Infor ION Development Guide–Cloud Edition

2020-x

Publication Information

Release: Infor ION 12.0.x
Publication Date: May 7, 2020
Document code: ion_12.0.x_ioncedg_en-us

Contents

About this guide

This guide explains how to adopt ION for new applications that you want to connect to ION.

The guide also explains how you can add metadata for your own documents or for extensions on standard documents.

Intended audience

The document is intended for this audience:

- System Administrators
- Business Process Administrators
- Business Analysts
- Database Administrators
- Application Administrators

Related documents

You can find the documents in the product documentation section of the Infor Support Portal, as described in "Contacting Infor".

- *Infor ION Desk User Guide*
- The "*Infor Ming.le administration*" section in the *Infor Ming.le Cloud Edition Online Help*
- *Infor ION API Administration Guide*

Contacting Infor

If you have questions about Infor products, go to Infor Concierge at https://concierge.infor.com/ and create a support incident.

The latest documentation is available from docs.infor.com or from the Infor Support Portal. To access documentation on the Infor Support Portal, select **Search > Browse Documentation**. We recommend that you check this portal periodically for updated documentation.

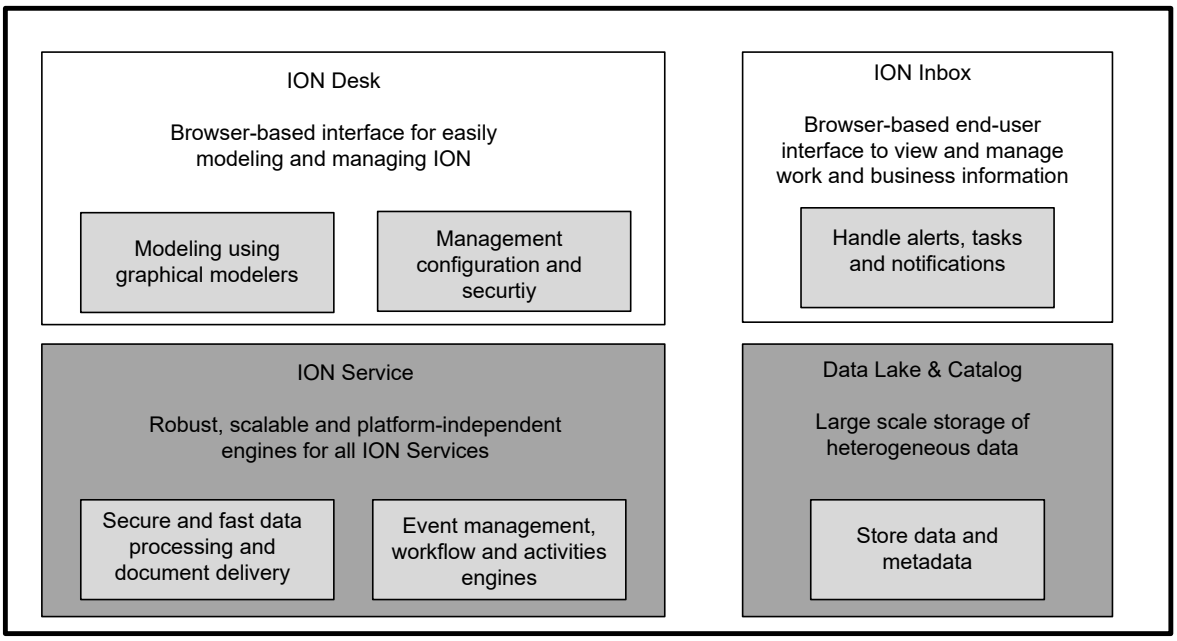
If you have comments about Infor documentation, contact documentation@infor.com.

Chapter 1: Introduction

ION is a new generation of business middleware that is lighter weight, less technically demanding to implement, and built on open standards.

In addition to connectivity with ION, you get workflow and business event monitoring in a single, consistent architecture. ION uses an event-driven architecture. It can pro-actively push data, work activities, and exception notifications to users. The ION Suite includes several powerful services to install and configure.

This diagram shows the ION Suite services:



ION Connect

With ION Connect you can establish connections between applications, which can either be Infor applications or third party applications. A set of connectors is available to connect many types of resources such as Infor applications, databases, message queues, or files. This varies from cloud or on premises. In ION Desk you can model document flows between applications. Such flows can represent a business process. But also more technical flows can be defined. For example, to map data

from a third party application to a standard business object document as used by an Infor application. You can also use filtering or content-based routing.

Workflow

In Workflow you can model business processes. A workflow can include tasks to be executed by a user, notifications to be sent to a user, decisions, parallel flows and loop backs. The modeling is done graphically. Workflows can be used to automate approval processes, and for other types of business processes. For example, a review flow consisting of several parallel tasks that are sent to multiple users to review the same document.

Event Management

Using Event Management you can monitor business events that are based on business rules. Users receive an alert when an exception occurs. For example, if a stock quantity is low, if a shipment is late, or if a contract must be renewed.

Pulse

With Pulse you can follow what happens in your organization. Either by following specific business documents or by following alerts, tasks or notifications.

Adopting ION

Adoption of ION is relevant in these situations:

- You want to integrate with other applications.
- You want to extend your application with Event Management, Workflow, or Pulse functionality.

To adopt ION these steps are required:

1. Preparation

   Getting the requirements clear is a major factor for success. What is the business case? In particular to identify business case of integration, system(s) to integrate with, data to exchange between systems, and mapping from BODs to data in involved applications.

   Event Management or Workflow capabilities, or both, can play an important role in adoption consideration.

   An introduction to ION as a product, see the *Infor ION Desk User Guide*.

   Additionally, it is important to understand some basic terms that are relevant for ION.

   See BODs and messages on page 12.

2. Connect to ION.

   Enable the application to connect to ION.

   See "Connecting to ION".

3. If required, adopt Event Management, Workflow, or Pulse.

See Adopting Event Management, Workflow, or Pulse on page 50.

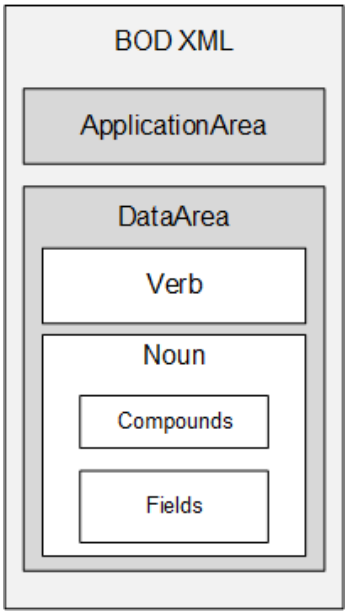
Chapter 2: BODs and messages

This section explains the terminology to adopt ION for an application.

BOD

A Business Object Document (BOD) is an XML document being a generic representation of a business object. A common language used for information integration. Infor have defined a set of standard BODs. At a high level, all BODs have some common characteristics. This standardization makes it easier to understand and use various BODs.

A BOD contains two parts: a noun and a verb.



Noun

A noun is a definition of a set of business data contained in a BOD. The noun represents the properties of one business object. Examples of nouns are SalesOrder, Item, and BusinessPartner. In ION Desk, a noun is called a document.

A BOD is data instance of one noun definition. A BOD message can contain multiple instances of the same noun definition.

Verb

The verb describes the action that is requested for the noun or indicates a response to an action.

A verb can:

- Announce that a business object is created, updated or deleted.
- Indicate a request to create, update or delete a business object.
- Provide a response to a request.
- Report an exception.

This table shows the request verbs supported by Infor:

Request verb	Description
Sync	A synchronization message containing changes that took place to a business object. A Sync message is sent by the owner of the data and can be delivered to any other application for which this information is relevant.
Process	A request to create a business object or to apply changes to an existing business object. A Process message is sent from any application to the application that owns the data. The owner will send an Acknowledge message in response to the Process request. The loaded document can be refused.
Get	A request to get the details for a business object. A Get message is sent to the owner of the data. The owner will send a Show message in response to the Get request.
Load	The Load verb is used when a document is created by an application that will not be the owner. A Load message is sent to the owner of the data. The loaded document cannot be refused.
Post	The Post verb is similar to the Process verb, but it does not trigger the creation of an Acknowledge message.
Update	The Update verb is used when data is changed by an application that does not own the data. The Update verb is similar to the Load verb, in the sense that it must be accepted by the owner. Namely, the Update message informs the owner of the data that an event took place and what data was changed by the event.

This table shows the response verbs supported by Infor:

Response verb	Description
Acknowledge	An Acknowledge response is sent in reply to a Process request. An Acknowledge response indicates whether the object to be processed was accepted, modified, or rejected.
Show	A Show response is sent in reply to a Get request.
Confirm	A Confirm verb is used when a failure happens. A Confirm verb is processed within ION and is not routed to any other application. The Confirm verb is used only for the BOD noun. The ConfirmBOD contains a copy of the original message, to enable an ION administrator to resubmit the same message after fixing the cause of the problem.

If you implement custom nouns, we recommend that you use these verb patterns as applicable:

- Sync
- Process and Acknowledge
- Get and Show

General concepts

The general concepts are explained here.

Tenant
A tenant is a hosting or software as a service (SaaS) concept where all the data of one tenant is always separated from all the data of other tenants. There is no cross sharing or viewing of data with other tenants. This concept requires all participants in the messaging to share the same identity for the same tenant. Therefore, a Tenant Id such as "infor" must have exactly the same meaning on every system in the messaging space. Tenants can also be used to separate data for a single on-premises customer, for example, separating data for a test environment from data from the production environment. Tenant Id is alphanumeric, maximum length is 22 characters.

Message Id
The Message Id is the unique identifier required for each message. The Message Id is used to detect duplication and to refer to the message in other messages, especially response documents and ConfirmBOD messages, and for logging. Message Id is alphanumeric, maximum length is 250 characters.

Logical Id
Logical Id provides a name for the instance of an application connected to ION. As the sender and or receiver of all messages, the application instance is identified by its Logical Id.

Applications must ensure that the Logical Id used in the message matches the Logical Id defined in ION.

Logical Ids are also used to drill back to an application in an Infor Ming.le environment.

The Logical Id has format: infor.[application or connection type].[Instance name in ION]. The ‘application or connection type’ can refer to Infor applications such as ‘syteline’, ‘eam’ or ’ln’, or refer to technology

connectors, such as ‘file’, ‘ws’ or ‘listener’. The 'instance name' is derived from the connection point name or message listener name. Logical Ids are alphanumeric, maximum length is 250 characters.

The only characters permitted in a Logical Id are the lowercase letters a-z, the dot(.), the digits 0-9, an underscore (_) and a dash (-).

Documentation Identification

Here is explained how to identify documentation.

Document ID
The ID of the document, is also available in the BOD, for example: SyncSalesOrder/DataArea/SalesOrder/Header/DocumentID/ID. As the unique identifier of an object Document ID is made up of several elements: the tenant, accounting entity, location, ID, and RevisionID.

Document ID is the simple ID for this object in its context, for example respective accounting entity, tenant etc. When in most cases, it is included in the BOD, Document Id (possibly along with Revision Id) is referenced in identifying the document in order to track the document in ION OneView, trigger event monitors, activation policies etc.

Document ID is alphanumeric, maximum length is 100 characters.

Revision ID
Revision ID sets the value that is used to keep multiple versions of the same document unique. An example of Revision ID: SyncSalesOrder/DataArea/SalesOrder/Header/DocumentID/RevisionID. Revision ID is alphanumeric, maximum length is 22 characters.

Variation ID
The variation ID of the document, when a Sync BOD is sent out from a system, a variationID is required. This variationID can be used at the receiving end to discover messages that are received out of sequence. For example: SyncSalesOrder/DataArea/SalesOrder/Head- er/DocumentID/ID/@variationID. Variation ID is numeric, the maximum value is 9223372036854775807.

Accounting Entity
Accounting Entity usually represents a legal or business entity, which owns its general ledger. Every single transaction only belongs to an Accounting Entity. Accounting Entity can also be defined as the owner of certain master data among the enterprise. Accounting Entity is alphanumeric, maximum length is 22 characters.

Location
A location is a physical place that is associated to a transaction. A location is owned by a single accounting entity, and may be used by multiple accounting entities. Location is alphanumeric, maximum length is 22 characters.

Message headers

Some header fields are identified as optional and some are mandatory in message headers.

Mandatory fields

Applications must guarantee the required fields are filled with correct values and format. Otherwise the message cannot be delivered or processed by consuming applications.

These header fields are required:

TenantID
The Tenant Id identifies the message as belonging to a specific tenant.

See the explanation in General concepts on page 14

MessageID
The Message Id is the unique identifier required for each message. See the explanation in General concepts.

BODType
The BOD Type header is used in ION Service to determine the verb and noun contained in the message. The verb and noun are used in routing algorithms. They are also used by client applications to filter messages from the inbound message queue. This header is case-sensitive and must follow the case of the verb and noun as used in the OAGIS schema. In the BOD Type, verb and noun are connected using a dot, that is: [verb].[noun]. For example, `Process.PurchaseOrder` or `Sync.Shipment`. We recommend that you use a noun name with maximum length of 30 characters. The verb does not exceed 11 characters. Acknowledge is the longest supported verb name. The maximum length of a BOD Type is 100 characters.

FromLogicalID
The Logical Id of the sending application. This Id is used to identify an application. Every application must have a unique identifier so that ION can direct documents to that application.

An example of a logical Id is:

```
lid://infor.eam.myeaminstance
```

For maximum length and other details, see the explanation on logical ID in General concepts on page 14.

Note: In case of network (tenant-tenant) connection.

When messages are transferred to a different tenant, the original source `FromLogicalID` is replaced with the logicalID of Network connection point on the target side.

ToLogicalID
The Logical Id of the destination application.

- Explicit routing:

  ION Service routes the message according to the Logical Id provided. This Logical Id must be valid for the specified routing rules.

  You must use explicit routing for BODs that have an Acknowledge or Show verb.

  Note: In case of network (tenant-tenant) connection.

When messages are transferred to a different tenant, the original `ToLogicalID` is replaced with the 'default' value on the target side.

- Implicit routing:

  The sending application is not aware who is interested in its message. ION Service routes the message according to specified routing rules. Therefore, the `ToLogicalId` header must be set to `lid://default`

  You must use implicit routing for BODs that have a Sync, Process, Load, Post, Update, Get, or Confirm verb.

For maximum length and other details, see the explanation on logical ID in General concepts on page 14.

Optional fields

The sending application must leave the optional header fields out or ensure they are filled correctly, consistent with the BOD XML contents.

In ION the header fields are not filled if they are missing or containing blank fields; but the data is transported. The receiving application can use the values if they are available, otherwise it can fall back to the BOD XML contents. The optional header fields are not used in all connectors. For example, the values of the header fields are not passed on to the stored procedure of a database connection point.

The optional header fields values cannot be null. Using `0` or a blank string `""` is allowed. When using an Oracle database for in-box and outbox, blank strings are treated as null. Consequently:

- When your application tries to write an optional header field with a blank string value to the outbox in Oracle, the insert action fails.
- When delivering a message to an in-box in Oracle, optional header fields with a blank string value are skipped in ION. This is to avoid failure of the message delivery.

Note: The data for the header fields is usually also available inside the BOD message. The sender is responsible for the consistency between the data in the header and the data inside the BOD. Header fields as set by the sender are not corrected in ION.

This list shows the header fields to identify the document that is included in the BOD.

- `AccountingEntity`
- `Location`
- `DocumentId`
- `RevisionId`
- `VariationId`

For the definition of each of these fields, see Documentation Identification on page 15.

Various header fields are available to support batch processing for large documents. In case of batch processing, the data can be sent in multiple BODs. The batch information is included inside the BOD, in the BODID element. Including it in the header can help the receiving application to handle the messages belonging to the same batch without opening the BOD messages.

Database connection point AnySQL type is populating batch headers.

This list shows the header fields to support batch processing for large documents:

- `BatchId`
- `BatchSequence`
- `BatchSize`
- `BatchRevision`
- `BatchAbortIndicator`

For details on these fields, see Sending messages in batch on page 46.

To further describe the message content, you can use these additional header fields:

- Instances

  Instances header is used to store count of object instances in the message. For example:

  - For a streaming JSON message, it indicates the number of instances in the stream.
  - For a Show BOD that includes multiple instances, it indicates the number of instances in the DataArea.
- Source

  Source header can be used to preserve information about the message source. In case read file action of File Connector is used, the source header is automatically populated with the source file name including file extension.
- Custom

  Custom headers can be used to preserve any other type of information that is not covered by other headers. You can use up to three custom headers per document. Each header must have the `Custom_` prefix.

Deprecated field

'Encoding' is an optional header field which is deprecated.

Details of Encoding see Documents encoding on page 27.

Chapter 3: Verbs and Verb Patterns

Standards are applicable to send your documents through ION and to enable all ION functionality for your document.

Functionality such as content-based routing and event monitoring. These standards are relevant for all documents, including custom documents.

Verbs

An important concept related to the BODs is the system of record. This is the application instance that owns a business object. A system of record can own all instances of a noun or it can own part of the instances.

Being the system of record or not determines the verbs to be used:

- Sync is sent from the system of record.
- Process, Get, Load or Update are requests that are sent to the system of record. For Process, the system of record will send an Acknowledge in reply. For Get, the system of record will send Show in reply.
- Confirm is a special type of verb. It is used for noun ‘BOD’, and it is sent when an error occurs in processing inbound BOD.

This table shows which verb to use for a specific goal:

Goal	Verb(s) to use
Publish changes on data owned by my application	Sync
Request changes on data owned by another application	Process/Acknowledge
Retrieve data owned by another application	Get/Show
Initial load, recovery	Get/Show (*)
Report an exception	Confirm

(*) This is the preferred approach. In theory you can use Show without Get, but then the sender must know the address (logical ID) of the receiver. An alternative is to use Sync. Especially when adding a new system of record. But when using this verb, event monitors or activation policies can be triggered.

This is what you do not want if Sync messages were already published before for the same data set. Because the messaging is asynchronous, the application that sends the process must have a way of handling the pending state until it receives the Acknowledge. For example, when requesting creation of a new item, the requesting application cannot use the item as if it were there already. In the meantime the application can use a specific status for the item such as ‘Pending’. Do not use other verbs. In addition to the listed verbs, ION supports using the Load and Update verbs. These are meant for integrations where data is loaded into an application where the application must avoid refusing the data. For example in an EDI scenario.

Action codes

The verb only indicates the action at a high level. It does not indicate whether a document is created, updated, or deleted. For such details, an action code can be included in the BOD. The action code is an attribute that is part of the verb section.

This table shows action codes for request verbs (except Get):

Action Code	Description
Add	A new business object instance is created.
Replace	A business object is changed and the complete business object is available in the message.
Change	A business object is changed and the message only contains the document ID and the changed properties. It is not recommended to use this action code.
Delete	A business object is deleted.

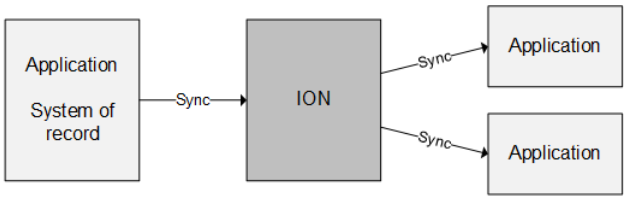
This table shows action codes for the Acknowledge response verb:

Action Code	Description
Accepted	The business object was created, changed or deleted in accordance with the Process request.
Modified	The business object was created or changed, but the resulting business object differs partly from the requested one.
Rejected	The creation, change or deletion was not done.

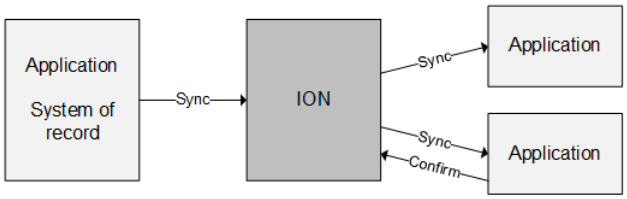
Note: For Sync the action code is the view of the system of record. That may not match the status of the receiving application. For example, SyncMyDocument with action code ‘Add’ and status ‘Draft’ is sent by application A. But it is not delivered to application B, because of a filter in the document flow. When application A sends SyncMyDocument with action code ‘Replace’ and status ‘Approved, then this will be the first document sent to application B. Application B has to add data, even though the action code is ‘Replace’.

Sync verb

Sync is sent from the system of record to anyone who is interested (‘broadcast’). It indicates that a business object was created, changed or deleted. The action code can be Add, Replace or Delete. Do not use action code Change. Use Delete only for tenant-level master data.



In case of failure for one of the recipients
(other recipients can accept the documents):



Publishing a Sync BOD

If you are the owner of a piece of business object data that can be represented in a BOD document. You can send Sync BODs to inform other parties about the current status. It is important to find the right balance when publishing Sync BODs. You do not want to publish too many BODs and not all changes to a document are relevant for the outside world. However, publishing less BODs can limit a customer when creating an integration or when using event management or workflow.

Take this into account:

1. When is my business object complete?

   In case of order entry, when the order header is saved but the lines are not added yet. It is not useful to publish a sync BOD for the order document. On the other hand, don’t be too late. For example, you can say “I will not publish this document until it has status ‘Approved’, because it is still being updated”. But this means that an event monitor cannot generate an alert until the document is Approved, which can be too late for corrective action.

2. Which status changes are relevant?

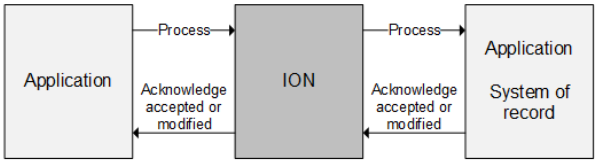
   It is hard to give general guidelines. What is important for one business document may not be important for another document. Changing a description or adding a note may not be relevant outside the context of your application. Changing the status or changing important data such as amounts or dates will usually be relevant.

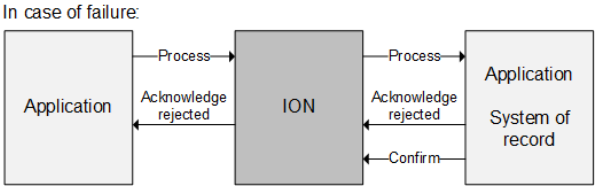
3. When is my business object at the end of its lifecycle?

Usually data is not deleted but archived when a business object is no longer current. This is important to sync out. It will inform other applications to not use the business object anymore.

Process and Acknowledge verbs

Process is a request sent to the system of record to add, change or delete a business object. Action codes for Process: Add, Change, Delete. Acknowledge is the reply sent back to the requestor. Action codes for Acknowledge: Accepted, Modified, Rejected.





Because the messaging is asynchronous, the application that sends the process must have some way of handling the pending state until it receives the Acknowledge. For example, when requesting creation of a new item, the requesting application cannot use the item as if it were there already. In the meantime the application can use a specific status for the item such as ‘Pending’.

When sending a `Process` BOD with an `Add` action code, the system requests a new record to be created in the Systems of Record (SOR), at this point the unique ID of the object is yet to be created in the SOR. Hence no Document Id must be populated in the `Process` BOD, but is populated in `Acknowledge` BOD when the record is successfully created in the SOR.

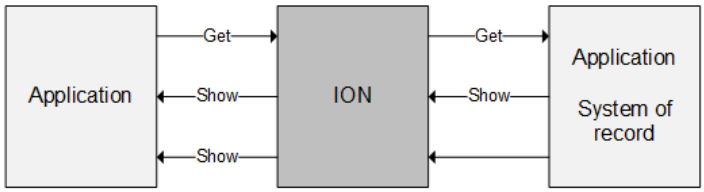
When the `Process` BOD gets action code `Change` the sending system requests a record to be either modified or deleted or to kick off a process related to the data object sent through the BOD. In this case Document Id, and Variation Id must still be populated to identify corresponding record in SOR.

Infor does not use action code `Delete` in Process BODs.

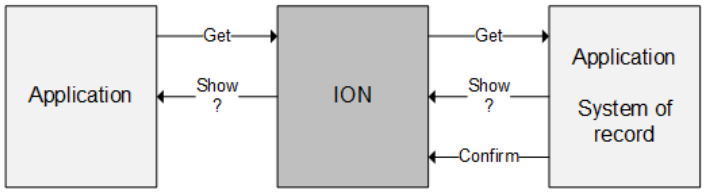
Get and Show verbs

A Get request is sent to the system of record to retrieve one or more business objects. A ‘query language’ kind is used to select the business object(s). The Show is the reply to a Get request. In Infor, the Show is the only BOD that can contain multiple nouns instances. A Show can also be sent without a Get

request. This can be used to push data for initial load or recovery. We do not recommend this, because it requires that the sender of the Show must know the applications that are interested in this message. No action codes are used in Get and Show.



In case of failure:



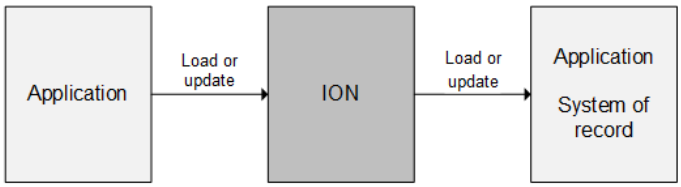
The Get verb is used to request data from a system of record (SOR). Either for:

- The purposes of an initial load or reload.
- To get specific instances of a document that the sending system typically does not receive.

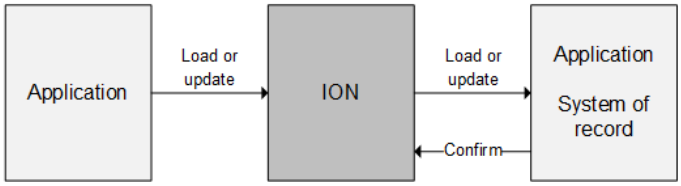
Include the Document Id if the Get BOD requests a specific object from the receiving system. Otherwise a selection criteria through expression element is used.

Load and Update verbs

Load and Update are used in special cases. The loaded documents cannot be refused by the system of record. These verbs are typically used for EDI. The Load verb is used primarily when a document is created by a trading partner and then passed into the SOR for the noun. Therefore in similar Process, there is no Document Id to specify in the BOD. Load is sent to the system of record to add an object. The action code will always be Add. Update is sent to the system of record to notify of a change. The action code will always be Replace.



In case of failure the system of record is not supposed to reject the document, but an error can occur:



Confirm BOD

A ConfirmBOD is sent by an application when processing any inbound BOD and something has gone wrong. A ConfirmBOD is always handled inside the ION Service, apart from ION it can never be sent to another application.

When to send Confirm BOD

When an exception that results in an inbound BOD not being processed occurs. The application must abort the transaction and send a ConfirmBOD.

Sending Confirm BOD from Process BOD

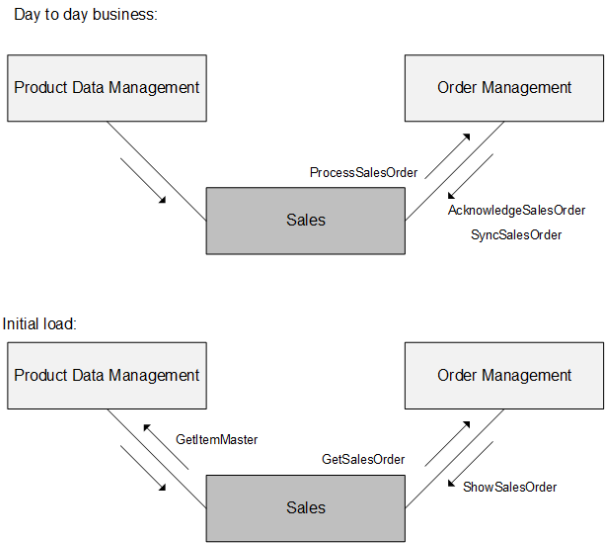
When sending a ConfirmBOD for a Process BOD, also send an Acknowledge with action code ‘Rejected’. Include a ReasonCode to explain the problem. The sender must be informed through the Acknowledge. In an Acknowledge BOD you must include the original ApplicationArea from the corresponding Process BOD. If you cannot get the original ApplicationArea, you can send a ConfirmBOD without an Acknowledge ‘Rejected’ BOD.

Example of Use verbs

Let us assume that three applications exist:

- Product Data Management: maintain product data (system of record for the ItemMaster noun).
- Order Management: handle sales orders (system of record for the SalesOrder noun).
- Sales: view the products and place orders

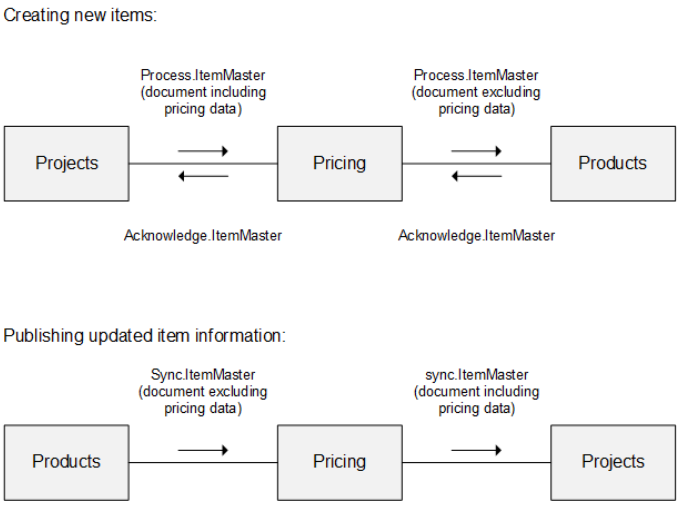
In this case, these BODs can be used to integrate these applications:



Fragmented data

Data for a single business object can be stored in multiple applications. But it is strongly advised that at any point in time one application is the owner of a specific piece of information.

If parts of the business object are owned by multiple applications, an enrichment or pipeline pattern can be used. For example, an ItemMaster business object is partly owned by a Product application and partly by a Pricing application. It is assumed that a third application exists called Projects. This application requests items to be created (using Process BODs) and must receive messages if an item is changed (using Sync BODs). See this diagram for the flows you can use:



For more information about verb definition and context to use, see the Infor Messaging Standards in Message contents on page 27.

Network connection

Reply verb patterns, such as Process and Acknowledge, Get and Show, Load and Update, have some limitations when the source and target applications are in different tenants.

In this scenario, the target application does not receive the original source `FromLogicalID` in the message header and is therefore not able to set `ToLogicalID` directly to the source application. The reply message is delivered to the source tenant with `ToLogicalID` 'default'. Then the automatic fallback mechanism ensures that the message is delivered to the source application. If there are more applications producing the same request message in the source tenant, each reply message is delivered to all of them.

Chapter 4: Message contents

Miscellaneous Infor messaging standards are supported by ION which applications must follow.

Noun references

Many nouns are referenced inside other nouns, both master data and transactional nouns. Referenced information can be found inside a BOD in these ways:

- Specific references used for transactional data, end with the word `Reference` and have the noun name at the beginning.
- The generic reference of transactional data through the `DocumentReference` element that uses the `type` attribute which contains the noun name of the referenced object.
- Used with master data which uses the name of the object as the reference and includes additional information with the reference information.

Documents encoding

An Infor standard is that all XML documents use the UTF-8 encoding. Therefore, there is no need to set this key. The default is UTF-8.

All readers and writers to the database tables that are used by the Infor Application Connector must use the exact same encoding mechanisms. When writing a message to the `COR_OUTBOX_ENTRY` table, serialize your XML string to the C_XML column as a UTF-8 byte stream. Similarly, when reading the `COR_INBOX_ENTRY` table. You extract your XML string from the C_XML column by converting the bytes as a UTF-8 byte stream.

If you do not use the UTF-8 decoding when reading the XML from the table, and the XML contains characters that require two bytes, the XML can be invalid. Also the formatting is affected. A worst case scenario is that you cannot parse the XML document.

Date and time

When working on ION integrated date and time format fields, follow these guidelines:

- Dates and times in BODs are in UTC time.
- They must be represented in a common format, using the capital letter Z at the end of the value. For example: 2009-08-13T15:30Z.

Chapter 5: Connecting to ION

Applications in the cloud or on premises can be connected to ION using the Infor Application Connector.

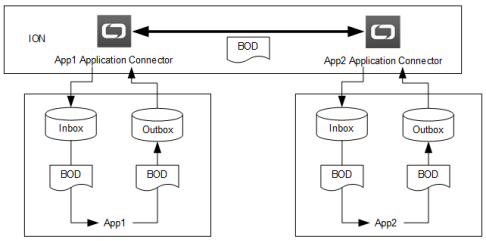
The application is truly event-driven. BODs (Business Object Documents) are used as the standard interface. Using BODs in combination with the in-box/outbox avoids unwanted dependencies and makes the integration robust. In ION, hardly any configuration is required to connect your application.

On the other hand, the application must be changed or extended to enable the application to publish and receive BODs. Consequently this is less suitable for customer-specific integration of legacy on-premises applications. Alternatives in that case are third-party connectors such as File Connector.

Infor Application Connector

The Infor Application Connector makes use of inbox/outbox tables in your applications. ION will post messages into your inbox and will pick up messages from your outbox.

This diagram shows an example of two applications exchanging messages using inbox and outbox tables:



To use the Infor Application Connector:

1. Create the inbox/outbox tables. Inbox and outbox tables must be created inside your application. The tables will be accessed by ION through JDBC. SQL scripts are available from ION Desk.
2. Implement the sending and receiving of the BODs. When a BOD must be published, create the BOD XML and determine the correct values for the header fields. Then insert the data into the Outbox tables. When a BOD is received in the Inbox tables, pick up the BOD XML and based on the content insert or update you application data.

For details, see Using the Infor Application Connector on page 38.

Using third-party connectors

Alternative connector

In addition to Infor Application Connector, File Connector is provided as alternative where files are used to send or receive messages. For details on these connectors, see the *Infor ION Desk User Guide*.

Justification of file connector

Using the Infor Application Connector is preferred, because it is decoupled and event-driven, and the application can validate the correctness and consistency of incoming data. From a modeling and management perspective, the Infor Application Connector also is the best choice, because the modeling is very simple and the management in ION Desk is the richest for this connector.

If you cannot use the Infor Application Connector, you can consider to use File Connector. File Connector is suitable for legacy applications that only support a file-based integration, and it can be event-driven. However the application at the other side must be in a position to create files in a format that ION can process, or read files as provided by ION. The data must be transformed to a BOD (XML) message. Multilevel data (header and lines) can be formatted in multiple ways in one or more files, while the File Connector does not support all options. There is a risk of delivering the same message twice in case of interruptions or timeouts.

Chapter 6: Infor Application Connector (IMS)

This section explains the adoption procedures that applications must fulfill to integrate with ION through the ION Messaging Service, (IMS), connection point.

IMS is a connector that allows applications to integrate with ION through REST/JSON APIs. Unlike the IO Box connector, IMS does not require direct access to an application's database. Instead, IMS communicates through the secured https protocol, through OAuth 1.0, or through ION API for authentication. Therefore, IMS is a loosely coupled connector that makes integrations easier.

IMS specifications include well defined API methods. These methods are implemented by ION and must be implemented by the concerned application. After this, they can push messages to each other through the APIs.

IMS can send and receive multiple message requests in parallel. Therefore, sequence of message transport is not guaranteed when using the IMS connector. If sequencing is important, you must use, for example, a `VariationID`.

The current IMS protocol supports three versions: v1, v2, and v3. March 2020, IMS protocol v3 was introduced. New adopters should use the v3 protocol. The v1 and v2 protocols are not described in this document.

Encoding

For transport performance reasons, especially where it involves larger messages, you can compress the message before sending. In that case you can use the DEFLATE or GZIP encoding. ION ensures the message is decompressed before it is delivered to the receiving application.

ION internals are DEFLATE-optimized. Therefore, we recommend the use of DEFLATE over GZIP.

IMS v3 introduction

These are the reasons to introduce an IMS v3 protocol:

- When ION IMS sends a `multipartMessage` with v3, the message is sent in a format that is compliant with swagger v2. This can be a breaking change for some applications. Therefore, a new IMS protocol version is required.
- Introduce a new method: `acceptedDocuments`. Applications can call this method to check which documentsIMS is prepared to receive.

This is a new method in ION. Therefore, a new protocol version is required. So an application can check whether ION is prepared for this method. This is relevant for on-premises deployments.

With that the integration flow with ION changes.

- ION Desk exposes APIs to configure the IMS connection point. These APIs are a good replacement for the `discovery` method.

  Extend the IMS endpoint definition with this property: `discovery=false`. The same can also be set as part of the `protocol` method.

  ION Desk supports documents that are defined through the UI / API rather than through the `discovery` method.

  You can now add documents in the same way as for a regular connection point.

Compared to IMS v2, IMS v3 offers these features:

1. Consistent implementation for `multipartMessage` method.

   In IMS v2, a `multipartMessage` that was sent by ION itself deviated from the swagger definition. This issue has been fixed. It was the main reason to introduce version v3 because this might be a compatibility issue for some adopters.

2. The `test` method is replaced with the `acceptedDocuments` method.

   You could use the v2 `test` method to check for a specific document if ION was prepared to receive it. The `acceptedDocuments` method lists the documents that IMS accepts for a specific connection point (logical-id).

3. More granular HTTP status codes.

   `multipartMessage` v2 throws a 412 error, both in case of configuration errors and malformatted messages. In v3 this is improved as follows:

   - A 412 error is thrown for configuration issues, such as an inactive connection point.
   - A 400 error is thrown for bad messages, for example because of a missing property.

   This way, adopters can handle the response more specifically.

4. The `protocol` method has an extra option to define whether the `discovery` method is available.

   Traditionally, the two-way (bidirectional) IMS integration required that applications used the `discovery` method to list the documents they can exchange. Nowadays, ION offers APIs to configure ION Desk models. Applications can call these ION Desk APIs to add and adjust the documents that are configured on a connection point. Therefore, the `discovery` method is optional. Applications can use the `hasDiscovery` property to indicate whether the `discovery` method is implemented by them. The default value is `true`.

5. Include `documentName` and `logicalId` as HTTP parameters.

   When ION sends a `multipartMessage`, the message includes the `documentName` and `logicalId` HTTP parameters. This way, an application can route the message internally without having to parse the actual message.

Guidelines for application teams that switch from v2 to v3

For a general understanding of the IMS interaction, see IMS interaction on page 34.

If your application currently supports v2, you must complete at least these steps to support v3:

1. Adjust: When calling the ION `versions` method, check whether ION supports at least v3.
2. If you still use the IMS `message` method:
   a. Adopt the `v3/multipartMessage` method.
   b. Adjust your `protocol` response to return `"messageMethod" : "multipartMessage"`.
3. Decide whether you are going to continue to populate documents through the `discovery` method. Alternatively, you can configure documents by pushing them to ION by calling the appropriate ION Desk API methods.

   If you plan to switch to using the ION Desk APIs, you must add this property in the protocol response: `"hasDiscovery" : false`
4. In IMS v2, when ION sends a `multipartMessage`, it actually deviates from the `multipartMessage` as defined in IMS swagger.

   With IMS v3 that issue is fixed, but as a consequence you receive the `multipartMessage` in a different format from ION.

   The differences are:
   a. The body for the payload now has a fixed `Content-Disposition: form-data;name="MessagePayload"`. With v2 ION would send the `Content-Disposition` with the same name as used in the `Content-Disposition` of the parameter body.
   b. The body for the parameters now has a fixed `Content-Disposition: form-data;name="ParameterRequest"`. With v2 ION would send the `Content-Disposition` with the same name as used in the `Content-Disposition` of the payload body.
5. After successfully processing a `multipartMessage` v3 request, ION sends an HTTP status code 202 (Accepted) instead of 201 (Created). We recommend that you handle any 2xx HTTP status response as a positive response. If you check specifically for 201, you must change your implementation.
6. Based on the `multipartMessage` HTTP status response that is provided by ION, you now can better identify these situations:
   - The method was called incorrectly: HTTP status response 400
   - The request itself is correct but the ION side is not yet configured to receive the message: HTTP status response 412
7. In IMS v3, the `acceptedDocuments` method is introduced as a replacement for the `test` method. With a single method you now can retrieve all documents that are accepted by a connection point in ION.
8. When ION sends a v3 `multipartMessage`, it includes the `documentName` and `logicalId` as HTTP parameters. Therefore, you can use these parameters to route the incoming request instead of having to parse the parameter body.

IMS interaction

Application sends a message to ION

Initialization phase: Prepare for sending messages

These methods are used by your application to get information and verify whether message processing is enabled:

1. `GET <Application>/service/ping`

   Checks whether ION can reach your application and checks whether ION is authorized for your application.

   Note: Although the ION ping still returns a body, ignore that body. That body response is deprecated and will no longer be sent when API v1 is no longer supported.
2. `GET <ION>/service/versions`

   Checks whether the ION version is v3 or later.
3. `GET <ION>/service/v3/<logicalid>/acceptedDocuments`

   Gets, from ION, the list of documents that ION is prepared to receive from the application (lid).

   HTTP status code 200: returns a list of documents that ION accepts to receive. If the connection point is inactive or does not exist, that is, not entitled, an HTTP status code 412 is returned.
4. If a document is missing:

   In ION Desk, open the connection point model and add the document to be 'sent by application'.

   Return to step 3.
5. Now the sending of messages can start for document types that are accepted by ION.

Note: For ION to accept a message, at least these conditions must be met:

- If IMS is called directly:
  - The OAuth credentials are correct, else HTTP status code 401 is returned.
  - OAuth principle is authorized for IMS, else HTTP status code 403 is returned.
- If IMS is called through ION API: the client-Id of the ION API-authorized App is configured in the IMS through ION API connection point.
- The `fromLogicalId` matches with the `logical-id` that is related to this OAuth principle. Else HTTP status code 403 is returned.
- The connection point is active and the document type is defined on the documents tab of the connection point.

Send a message

This section describes the sequence when sending a message to ION.

Only start this cycle after the 'prepare for sending messages' procedure. In other words: only send messages that are accepted by ION CE.

1. `GET <ION>/service/ping`

If this is successful, then continue. Otherwise retry and raise errors.

For the retry it is required to increase the delay between retries if the problem exists longer. Consider to aggregate the ERROR message to prevent log flooding.

If the returned error is a 401 or 403, then stop sending messages. Instead, take corrective action, for example, by getting a correct OAuth keypair from ION.

2 `GET <ION>/service/versions`

Check whether the ION version is v3 or later.

3 `POST <ION>/service/v3/multipartMessage`

`Http header X-TenantId`

If sending the message fails, based on the error code:

a HTTP status code 401, 403, 503, etc.: Return to the ping, step 1.

b HTTP status code 400: Raise an error to the admin that an invalid message was sent. Stop processing until the issue is resolved or skip this message and continue with the next.

c HTTP status code 412: Return to the initialization phase: Prepare for sending messages.

If the message was sent successfully, then continue with the next message, step 3.

ION sends a message to an application

Prepare your application to receive IMS requests from ION

Ensure your application exposes these methods, which are used by ION to get information and verify whether message processing is enabled:

- `GET <Application>/service/ping`

  Checks whether ION can connect to your application and checks whether ION is authorized for your application.

- `GET <Application>/service/protocol`

  Retrieves the expected IMS API version and protocol parameters from the application.

- `POST <Application>/service/v2/discovery`

  `Http header X-TenantId`

  Called during modeling of the connection point in ION Desk. Retrieves, from the application, the documents that are supported to be exchanged.

  Implementation is optional: if not implemented, ensure that the `hasDiscovery` property in the protocol method response is false.

- `POST <Application>/service/v2/multipartMessage`

  `Http header X-TenantId`

  Sends the actual message.

Configuration

Ensure an Infor application (IMS) connection point is configured in ION. This connection point must be used in a document flow. Ensure this document flow is activated.

Testing the connectivity

To test connectivity from ION to your application, you can click the **Test** button in the ION Desk connection point model.

Methods to verify connectivity and configuration

These methods are used by ION to verify connectivity and configuration:

- `GET <Application>/service/ping`

  Checks whether ION can reach your application and checks whether ION is authorized for your application.

- `GET <Application>/service/protocol`

  Retrieves the expected IMS API version and protocol parameters from the application.

  If this method fails, ION calls the `GET <Application>/service/versions` method and continues with v1. The application is probably on v1.

  If an ION Desk user opens the **Documents** tab on the IMS connection point screen, ION calls the `ping` and `protocol` methods. Based on the protocol response, ION enables or disables the **Discovery** option on the **Documents** tab.

Methods that are executed if "The IMS End point has Discovery" is true and the application IMS protocol version is v3

The `discovery` method can only be called if "`the IMS End point has Discovery`" is true. When ION Desk calls the `discovery` method, these methods are executed by ION towards the application:

- `GET <Application>/service/ping`

  Checks whether ION can reach your application and checks whether ION is authorized for your application.

- `GET <Application>/service/protocol`

  Retrieves the expected IMS API version and protocol parameters from the application.

  If this method fails, ION calls the `GET <Application>/service/versions` method and falls back to v1 if required.

- If the identified version is v3, ION calls the `POST <Application>/service/v3/discovery` method.

  `Http header X-TenantId`

  Retrieves, from the application, the documents that are supported to be exchanged.

ION sends a message, when application IMS protocol version is v3

ION executes these methods in the given sequence when a message is being sent:

1. `GET <Application>/service/ping`

   If this method is successful, then ION continues. Otherwise a retry is performed and errors are raised.

   If the retry remains to fail, ION slows down the pace of the retry.

2. `GET <Application>/service/protocol`

   Retrieves the IMS protocol parameters from the application.

   If this method fails, ION calls the `GET <Application>/service/versions` method and expects version v1.

3. `POST <Application>/service/v3/multipartMessage`

   `Http header X-TenantId`

   If sending the message fails, then, based on the error code, ION performs one of these actions:

   - Retry, that is, return to step 1.
   - Raise an ERROR for the specific message and continue with the next message.

4. ION continues with the next message, step 3.

Chapter 7: Using the Infor Application Connector

This section explains the adoption procedures that applications must fulfill to integrate to ION using the Infor Application Connector.

You must have some specific knowledge about these topics:

- Application connection points
- In-box and outbox tables
- Removing messages from the in-box and outbox tables

Application connection points

To connect an application to ION, the application owner must define which BODs that application can send and receive. To define the BODs for an application, you must create an application connection point. You can export connection points to an XML file. When you export the connection point without properties, you can use it as a template for the application.

See the *Infor ION Desk User Guide* on how to create and export connection points.

Inbox and outbox tables

All applications must add some of these new tables to their existing database so that the application can read and write the tables in the same transaction as their business logic:

- COR_OUTBOX_ENTRY
- COR_OUTBOX_HEADERS
- COR_INBOX_ENTRY
- COR_INBOX_HEADERS
- ESB_INBOUND_DUPLICATE

Caution: When an application adds a new message to the COR_OUTBOX_ENTRY and COR_OUTBOX_HEADER tables, the inserts must be performed within the same transaction. Inserts that are not performed within the same transaction cause data corruption.

To download the scripts to create those tables:

1. Start ION Desk.
2. Select **Configure > ION Service**.
   The **Configure ION Service** page is displayed.
3. Click the **Configuration Files** tab and then click **Download Scripts to create I/O Box**.
4. Specify a file name and click **Save**. A zip file is downloaded.
5. Extract the zip file and use the db vendor-specific sql files that are applicable to you. For Unicode-compatible BOD header fields, use the scripts available with the _unicode suffix.

Note: The script might also generate a COR_PROPERTY table. This table is included for future use, so you can safely ignore it.

COR_OUTBOX_ENTRY

This table shows the COR_OUTBOX_ENTRY table API:

COR_OUTBOX_ENTRY	Description
C_ID	The row's primary key - all of the provided database schemas have this set as auto-increment.
C_XML	The message that you are sending. The message must be encoded as described below.
C_TENANT_ID	The Tenant Id identifies the message as belonging to a specific tenant. A tenant is a hosting or software as a service (SaaS) concept where all the data for one tenant is always separated from all the data of other tenants. There is no cross-sharing or viewing of data with other tenants. This concept requires all the participants in the messaging to share the same identity for the same tenant. Therefore, a Tenant ID of " infor " must have exactly the same meaning on every system in the messaging space.
C_LOGICAL_ID	This field is added since ION 11.1.2. It is added by running the scripts present in the '3.0' folder in your inbox/outbox. If present, then this field must contain the value of the 'from logical id' of the application that publishes the BOD and it must be populated with the ‘lid://’ prefix.
C_MESSAGE_PRIORITY	Messages with a higher priority are sent before messages with a lower priority. You can set the priority from 0 to 9, 9 being the highest priority. High priority messages should be limited - most messages should be set at 4.

COR_OUTBOX_ENTRY	Description
C_CREATED_DATE_TIME	The date and time the message was inserted into the outbox table. You must specify the time in the variable in UTC format.
C_WAS_PROCESSED	Users should never provide a value for this column. The column is used by ION Service to determine whether a message has been sent. Unprocessed messages are marked as 0; processed messages are marked as 1.

ION Service removes all the processed messages older than the number of hours specified in the cleanup advanced properties.

COR_OUTBOX_HEADERS

This table shows the COR_OUTBOX_HEADERS table API:

COR_OUTBOX_HEADERS	Description
C_ID	The row's primary key - all of the provided database schemas have this set as auto-increment.
C_OUTBOX_ID	Used to join the headers to the message's COR_OUTBOX_ENTRY row. As such, this value should be the same as the message's COR_OUTBOX_ENTRY C_ID column.
C_HEADER_KEY	The key used to describe the type of header. For valid keys, see "Message headers".
C_HEADER_VALUE	The header's value.

If any of the required headers are not provided, ION Service creates a Confirm BOD for that message.

For the headers to be used, see Message headers on page 16.

COR_INBOX_ENTRY

This table shows the COR_INBOX_ENTRY table API:

COR_INBOX_ENTRY	Description
C_ID	The row's primary key - all of the provided database schemas have this set as auto-increment.

COR_INBOX_ENTRY	Description
C_XML	The message you are receiving. The message must be encoded as described.
C_TENANT_ID	The Tenant ID identifies the message as belonging to a specific tenant. A tenant is a hosting or software as a service (SaaS) concept where all the data of one tenant is always separated from all the data of other tenants. There is no cross-sharing or viewing of data with other tenants. This concept requires all participants in the messaging to share the same identity for the same tenant. Therefore, a Tenant ID of " infor " must have exactly the same meaning on every system in the messaging space.
C_LOGICAL_ID	This field is added since ION 11.1.2. It is added by running the scripts present in the '3.0' folder in your inbox/outbox. If present, then this field contains the 'To logical id' value of the application to which the BOD is delivered.
C_MESSAGE_PRIORITY	Message priority as provided by the application that sent the message.
C_CREATED_DATE_TIME	The date and time the message was inserted into the inbox table. Date and time are in UTC format.
C_WAS_PROCESSED	ION Service always sets this to 0. It is the application's responsibility to remove processed messages.

COR_INBOX_HEADERS

This table shows the COR_ INBOX _HEADERS table API:

COR_ INBOX _HEADERS	Description
C_ID	The row's primary key - all of the provided database schemas have this set as auto-increment.
C_ INBOX _ID	Used to join the headers to the message's COR_ INBOX _ENTRY row. As such, this value should be the same as the message's COR_ INBOX _ENTRY C_ID column.
C_HEADER_KEY	The key used to describe the type of header. For valid keys, see "Message headers".

COR_ INBOX _HEADERS	Description
C_HEADER_VALUE	The header's value.

Incoming messages are placed in the COR_INBOX_ENTRY and COR_INBOX_HEADERS tables. All rows are inserted in the same transaction.

ESB_INBOUND_DUPLICATE

This table is used by ION to maintain the unique Message IDs. ION uses this table to reject duplicate messages in the ION Service. Applications should not use this table.

Removing messages from the inbox and outbox tables

ION Service removes messages from the COR_OUTBOX_ENTRY and COR_OUTBOX_HEADERS tables. Removing these messages is achieved in these ways:

- The message is deleted after it is successfully sent by ION Service.
- The message is deleted if it has been successfully sent and is older than XX hours. ION Service checks for expired messages when it is started, and then checks every hour.

By default, the second option is used. By not deleting the messages immediately, the Manage tab in ION Desk monitors the COR_OUTBOX_ENTRY table and reports the number of processed and unprocessed messages. To change this behavior, set the application polling property within ION Desk:

```
Delete Processed Messages=true
```

Therefore, ION Service deletes the message after it is sent.

Caution: ION Service does not remove messages from the COR_INBOX_ENTRY/COR_INBOX_HEADERS tables. The application must provide the code to clean up these tables. If the tables are not cleaned up, the file system of the database server can get full.

Polling Message Preference

When you implement ION integration you can decide to use one of these options:

- single inbox/outbox shared by multiple sites (represented by multiple Logical Ids)
- single inbox/outbox shared by multiple tenants in the Cloud.

A ‘Message processing preference in I/O box’ setting is available in ION Desk Connection Point Advanced Settings, to cater for two requirements.

Single I/O Box for Multi-tenant

In a multi-tenant environment a single set of Inbox and Outbox tables of the application can be shared by multiple tenant instances. In such situations an application connection point must process messages belonging to its own tenant.

To setup Single I/O Box for Multi-tenant:

1. Go to **Connection Point definitionConnectionAdvancedMessage processing preference in I/O box**.
2. Select the **by Tenant** option to be **`true`**.
3. When publishing a BOD from your application, specify this information:
   - The `C_TENANT_ID` value of the `COR_OUTBOX_ENTRY` table.
   - The Tenant Id key in the `COR_OUTBOX_HEADERS` table with the correct tenant value. It must match with the Tenant value specified in the connection point.

Ensure you do not have more than one connection point from the same tenant sharing the same Inbox and outbox tables.

The Tenant value is matched in ION in a case-sensitive manner. If the tenant value is blank in the connection point, the default tenant value of 'INFOR' is assigned. To avoid inconsistencies in tenant processing, define the Tenant Id according to the Infor standards.

Duplicate detection of messages is not enforced by ION. Under uncommon circumstances such as an incomplete message processing, this results in delivering the same message twice to an inbox. The application must be prepared to receive duplicate messages.

Single I/O Box for Multi-Logical Ids

This addresses requirements from applications which have multiple sites where an individual connection point is defined per site in ION. These applications use the same Inbox and Outbox table between them. In such situations, messages based on Tenant and based on Logical Id must be processed.

To setup Single I/O Box for Multi-Logical Ids:

1. Upgrade the I/O Box to version 3.0. Go to the 3.0 folder in your I/O box and run the I/O box script `<db>_upgrade.sql`.

   The downloaded zip file contains these folders: 1.0 and 3.0. The folder 1.0 contains the scripts to create a base I/O box for the standard databases. This includes: MS SQL server, Oracle, DB2, DB2400 and MySQL.

   The 3.0 folder contains the scripts required to prepare your I/O box to support multiple logical Ids sharing the same I/O box. Run the scripts in the 3.0 folder to create I/O Box tables in your

application. For Unicode compatible BOD header fields, you can use the scripts available in the Unicode folder.

2 Specify the message processing preference to be **by logical Id** in each connection point defined for your application.

Go to **Connection Point definition > Connection > Advanced > Message processing preference in I/O box**.

Select **by Logical ID** to be true and select **by Tenant** to be true.

Before you proceed, check if the column called `C_LOGICAL_ID` exists in both `COR_OUTBOX_ENTRY` and `COR_INBOX_ENTRY`.

3 When you publish BODs from your application, ensure that the correct values are specified in these columns:

- `COR_OUTBOX_ENTRY` table, specify `C_LOGICAL_ID` with your actual Logical Id value.
- `COR_OUTBOX_HEADERS` table, specify the `FromLogicalId` key with the correct Logical Id value.
- Both Logical Ids must match the Logical Id value specified in the connection point.

Note: The Tenant and the Logical Id value are matched in a case-sensitive manner. To avoid inconsistency in tenant processing, specify the Tenant Id and the Logical Id according to the Infor standards.

When these properties are not selected, you must ensure that each Infor application connection point uses its own Inbox and Outbox tables. This will be guaranteed by using different URLs. If you use the same URL in multiple connection points, then use different users. Ensure the users are linked to different database schemas in the database management system.

Chapter 8: ION Connecting Considerations

There are some considerations for applications to take into account when implementing integration through ION.

Handling transactions

Ensure the publishing and processing of the BODs is done in such a way that no data is lost. Publish the BODs inside the application transaction that inserts or changes the corresponding business data. Or, if publishing is done offline, have another mechanism in place that ensures no BODs are lost.

Because ION is event driven you must consider how to publish historical data before you enable ION integration with your system. Data consistency must be honored. When handling an incoming BOD, set the status in the Inbox to ‘1’ (processed) in the same transaction as the database updates that are done while processing the BOD.

Message sequence

Delivering messages in sequence is not guaranteed. Only delivery is guaranteed (at least once). In some cases the sequence of receiving is the same as the sequence of sending, but you cannot rely on this. Many factors can impact the sequence, such as:

- Parallel processing (multi-threading).
- Documents traffic.
- Intermediate steps in the process, such as content-based routing, filtering or mapping.

Take these situations into account:

1. Message Delivered with Delay or Out of Sequence sometimes due to network or BOD traffic BODs are not always delivered on time or in sequence. Especially in case they reference to each other. For example; a `Sync.ItemMaster` BOD reaches the application later than a `Sync.PurchaseOrder`. The application may well send out a Confirm BOD for the `Sync.PurchaseOder` BOD. Or is highly recommended to set a retry mechanism to have a few tempts based on intervals to sort out the message delay or out of sequence issue.

2 Message Delivered Out of Sequence for Different Verbs of the Same Object. It happens sometimes when an application awaits the Acknowledge BOD from the SOR for the object via Process BOD. The Sync BOD of the same object from the SOR arrives prior to any Acknowledge BOD. This is similar to point 1, caused by BOD traffic or network, and can be solved by a few times retry in turn. Therefore applications are recommended to include this into IONAdoption consideration.

3 Multiple updates from SOR on same document out of sequence Multiple changes on the same document may not arrive in the correct sequence. Use the Variation ID to check whether a Sync message is out of date. If the Variation ID of an incoming message is lower than a Variation ID you already processed for the same document type and document ID, then you must not overwrite the newer information you already have.

4 Data with interdependency break down into a set of documents. This is the same as what is explained in "Sending messages in batch".

Duplicated messages

When publishing a message in an Infor Application Outbox, the message ID from the header will be checked. If a message with the same message ID was processed successfully before, the new message is ignored.

At the end of a flow, in exceptional cases the same message can be delivered twice (guaranteed delivery ‘at least once’). The receiving party must ignore a message if a message with the same message ID was processed before.

For that reason, ensure to use a unique message ID when sending a message to your outbox. Otherwise the message will be ignored. The message ID must be globally unique, so a sequence number generated by your application is not sufficient.

Sending documents in batch

Avoid sending large documents. Documents up to 5 MB are handled throughout ION.

Do not include large files inside the documents, for example, images or PDF files. Instead, make the files available in a document management system or another location. Include a reference (URL) in the document.

Sometimes you must divide a large file into multiple documents. For example, the first document can contain a header and the first 100 lines. The next document can contain the next 100 lines. Each document must be valid.

Batch message headers are used to indicate that the document is part of a batch. Batch fields should not be sent for single documents (batch size = 1).

For BODs, Batch fields are used to indicate that the BOD is part of a batch. The batch fields must be included inside the BOD XML, in the BODID, and can be included in the message header. The message header fields are optional. We recommend that you include them. It allows the receiver to handle the

batch without having to open each BOD to determine the correct sequence. Note that the total length for the BODID cannot exceed 255 characters.

The data types and maximum lengths of the header fields are specified in the table.

This table shows the fields to use:

Header Field Name	Name of element in BODID	Description
BatchId	`batchID`	The unique ID of the batch. The documents with this number must be processed sequentially on the receiving side. The **BatchId** is alphanumeric, maximum length is 250 characters.
BatchSequence	`batchSequence`	The sequence number of this document in the batch. This is required because the documents can arrive out of sequence. The BatchSequence is numeric, the maximum value is 9223372036854775807.
BatchSize	`batchSize`	The total number of documents in the batch. This can be unknown until the last document and omitted for the other documents. The **BatchSize** is numeric, the maximum value is 9223372036854775807.
BatchRevision	`batchRevision`	The revision number of the batch. This is used when one set of documents fails. The complete set can be resent using a new revision number. The revision number must be increasing, but does not have to be sequential. The **BatchRevision** is numeric, the maximum value is 9223372036854775807.
BatchAbortIndicator	`abortIndicator`	This indicator type attribute is set to true when a system that is sending a batch determines that it will not finish. The receiving system is notified that any documents received as a part of this batch must be discarded. The **BatchAbortIndicator** value is either ‘true’ or ‘false’.

For example, publishing daily balance updates through SourceSystemGLMovement BOD. Often you must split the BOD. It is assumed that the tenant is ‘acme’, the accounting entity is 10 and the location is 1. The subsequence BODIDs is one of these options:

- `Infor-nid:acme:10:1_A:0?SourceSystemGLMovement&verb=Sync&batchSequence=1&batchID=lid://infor.sunsystems.5:1`
- `Infor-nid:acme:10:1_A:0?SourceSystemGLMovement&verb=Sync&batchSequence=2&batchID=lid://infor.sunsystems.5:1`

- `Infor-nid:acme:10:1_A:0?SourceSystemGLMovement&verb=Sync&batchSequence=3&batchID=lid://infor.sunsystems.5:1&batchSize=3`

Publish historical data

Due to the synchronous nature of data in an event-oriented architecture, you can only keep data in sync after your application is integrated with ION.

You may need to consider how to share historical data with downstream systems that are interested to receive. This concerns the data which may not be amended in your system anymore. We recommend that you provide a facility to publish historical data when the application is ION-enabled. For example, driven by end users at time when required. This can happen more than once due to online offline deployment.

To keep the consistence of message ID and variation ID your system must publish BODs after historical data is published.

Message reprocessing

When BODs are delivered to your application inbox/outbox tables, it is the application's responsibility to consume or reject them due to circumstances. There are options to set intervals in ION Desk to clean up inbox/outbox tables. You can develop your own clean up scheme considering BODs volume and storage.

In case of processing inbound BODs fails for business data integrity, we recommend that you develop a user-driven retry mechanism. For example, workflow integration rather than mandating business users to submit the document for approval again. For technical reasons, it makes better sense if the system admin is able to resubmit the same BOD to get through. Same with consuming transaction BODs, ledger entry or purchase order quite often a retry is mandatory to sort out interdependency of BODs delivery.

When developing system automatic retry, it is important to retry at reasonable intervals and end the process if the problem is not solved. This is to avoid deadlock of your integration engine focusing on a few BODs and leaving the Inbound BODs queue too long to process.

Performance

Good performance is a key to success especially in an event driven integration architecture. Sensible plan and proof of concept testing with real business scenarios will eliminate substantial issues with customer implementation after releasing your ION integration.

Depending on the integration requirements we highly recommend that you take performance into account during the start and design phase.

Chapter 9: Adopting Event Management, Workflow, or Pulse

When the connection to ION is completed, you can adopt Event Management, Workflow or Pulse.

With Event Management you can generate alerts for business data that you publish. The alerts are created by an event monitor that checks the Sync BODs that are published against a user-defined rule.

Workflow enables you to execute workflows, either based on published business data or by explicitly starting a workflow. In Workflow you can implement decisions and bring tasks and notifications to users.

You can also create alerts, notifications, or tasks directly from your application. In that case you do not use Event Management or Workflow, but you directly request the Pulse engine to create an alert, notification, or task.

Alerts, notifications and tasks

A task is sent if the user is expected to execute a defined task. The user must complete the task in Infor Ming.le using the Tasks widget in the Homepages or in the Infor Ming.le Mobile application. To complete a task, the user specifies data or selects a specific action, such as 'Approve' or 'Reject', when closing the task.

An alert means a user must be notified of an exception. It indicates that something happened that is extraordinary or that is not in line with how the business should run. The user can decide whether to take action and then close the alert.

A notification is a message to one or more users 'for your information'.

Note: Some differences exist between the features that ION offers for tasks, alerts and notifications:

- Tasks have a priority (High, Medium or Low). Alerts and notifications do not have an explicit priority.
- The names (labels) of the data elements for an alert are not translatable. For tasks and notifications created from a workflow the labels are translatable.
- In alerts, drill-back links are automatically generated based on document references. For tasks and notifications that are created from a workflow, drill-back links can be configured. To configure these drill-back links, use drill-back views that are retrieved from the uploaded drill-back view sets.

When to use Pulse, Event Management and Workflow

Use Pulse if you want to create an alert, notification, or task directly. Your application is fully in control. The application logic decides whether and when to create an alert, notification, or task and also defines all aspects such as the data to be included and the user(s) who will receive the item. For tasks, your application also follows up if required when the task is completed.

Use Event Management if you need alerting, but you want to delegate the monitoring to ION. You do not have to change your application; the only requirement is that you publish Sync BODs for the business data owned in your application. You can define rules in ION event monitors. Customers using your application can adapt the rules to their needs or define new rules. ION monitors the BODs published from your application and creates alerts in Pulse when required.

Use Workflow if you want to model a business process (or allow your customers to model a business process) outside your application. Tasks and notifications are created automatically based on the modeled process. If required, the user interacts with your application based on the tasks that the user receives.

The following chapters describe:

- How to start workflows from an application.
- How to create alerts, tasks, or notifications from an application.

Chapter 10: Starting a workflow from an application

This section describes the details of the Workflow BOD and its Process/Acknowledge messages.

You can trigger workflow definitions in various ways. To trigger a workflow instance from an application, you can use these methods:

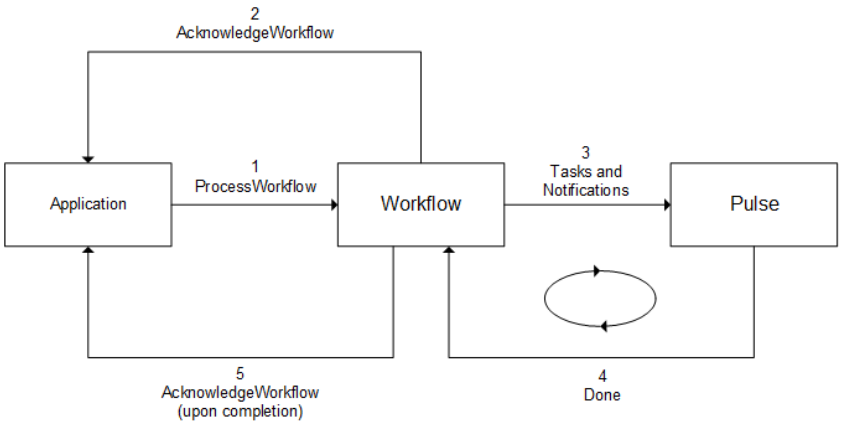
- Indirectly, by creating an activation policy or monitor that evaluates Sync BODs that are sent by this application. When a workflow instance that is started by this method completes and has output parameters. The activation policy creates a Process BOD of the same noun as the monitored Sync BOD. This Process BOD is sent to the originating application with the values resulting from the workflow execution.
- Directly, by sending a `ProcessWorkflow` BOD. When a workflow instance is started by this method, an `AcknowledgeWorkflow` is sent initially to inform that the initiation of the workflow was successful. When the workflow is completed, another `AcknowledgeWorkflow` BOD, which contains the values of the workflow output parameters, is sent back to the application.

For details about how to model and start a workflow and start workflow from activation policy and document flow, see the *Infor ION Desk User Guide*.

Starting a workflow through ProcessWorkflow

Create a workflow definition using the ION Desk workflow modeling. The actual process as modeled inside the workflow can be changed later, but your application depends on the interface of the workflow definition. The interface consists of the workflow name, the input parameters and the output parameters. To start a workflow, publish a `ProcessWorkflow` BOD, including the name of the workflow definition to be started and the values for the input parameters. This starts the workflow.

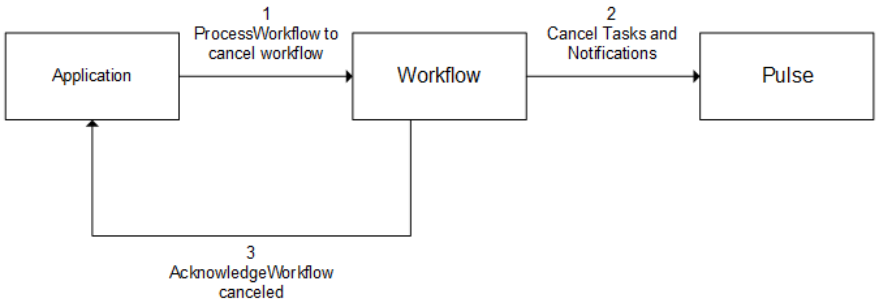
See this diagram:



You receive an `AcknowledgeWorkflow` BOD when the workflow is created. You receive also an `AcknowledgeWorkflow` BOD when the status of the workflow changes. For example, the workflow is completed or cancelled. If the workflow is completed, the BOD contains the values for the output parameters of the workflow. You can use the result in your application.

Note the difference between Workflow and Pulse BOD if you use Process verb. For Pulse BODs you receive only one `Acknowledge` BOD. For Workflow BOD you receive multiple `Acknowledge` BODs to be updated with different statues of the workflow task(s). If you started a workflow but the workflow is not relevant anymore, you can cancel it. To cancel a workflow, publish a `ProcessWorkflow` BOD.

See this diagram:



In this case you receive an `AcknowledgeWorkflow` BOD when the workflow is canceled. Specifications of how to create and cancel Workflow BOD to ION are discussed later.

To start a workflow, add this action code: `ProcessWorkflow/DataArea/Process/ActionCriteria/ActionExpression/@actionCode`.

This table shows the elements you can use in the noun instance, `ProcessWorkflow/DataArea/Workflow`:

Element	Note
WorkflowDefinitionCode	Required. This is the name of the workflow definition as modeled in ION.
Property/NameValue	Properties are required if the workflow model has input parameters. You must specify values for the input parameters that are required.

Do not use other elements, such as DocumentID and Status, when initiating a new workflow instance. These elements are determined by the Pulse engine.

The resulting AcknowledgeWorkflow BODs contain the actionCode with these possible values:

- "Accepted", when processing the request was successful.
- "Modified", to inform about an update in the workflow definition execution.
- "Rejected", if the request could not be processed.

This table shows the elements that are included in the `AcknowledgeWorkflow/Workflow` section if the actionCode is "Accepted":

Element	Note
DocumentID/ID	Unique identification of the workflow instance in this ION installation.
Status/Code	Value is "Initial" to indicate the workflow was started.
WorkflowDefinitionCode	The name of the workflow started.

This table shows the elements that are included in the `AcknowledgeWorkflow/Workflow` section if the actionCode is "Modified":

Element	Note
DocumentID/ID	Unique identification of the workflow instance in this ION installation.
Status/Code	Can be "Cancelled", "Failed", or "Completed".
Status/Reason	Available for Status/Code "Cancelled" or "Failed".
WorkflowDefinitionCode	The name of the workflow that was canceled, failed, or completed.
Property/NameValue	Only available if the Status/Code is "Completed" and the workflow has output parameters. The Properties contain the resulting values of the workflow output parameters.

This table shows the elements that are included in the `AcknowledgeWorkflow/Workflow` section if the actionCode is "Rejected":

Element	Note
Status/Code	Value is "Failed".
Status/Reason	The reason for failure.

Canceling a workflow through ProcessWorkflow

To cancel a workflow, the action code, `ProcessWorkflow/DataArea/Process/ActionCriteria/ActionExpression/@actionCode`, must be "Change".

This table shows the elements you can use in the noun instance, `ProcessWorkflow/DataArea/Workflow`:

Element	Note
DocumentID/ID	Required. Unique identification of the workflow instance that must be canceled.
WorkflowDefinitionCode	Required. This is the name of the workflow definition as modeled in ION.
Status/Code	Must be "Cancelled".

The resulting AcknowledgeWorkflow BOD contains the actionCode="Accepted" if cancelation was performed, or actionCode="Rejected" if the cancelation was not possible. The other elements included are similar to those described for AcknowledgeWorkflow BODs.

See Starting a workflow through ProcessWorkflow on page 52.

Workflow BOD details

The definitions of ProcessWorkflow and AcknowledgeWorkflow are available on this site:

http://schema.infor.com

This table shows the elements that exist in these documents:

Element	Note
DocumentID/ID	Unique identification of the workflow instance in this ION installation.
Status/Code	Can have these values: • "Initial": the request to start a new workflow instance was performed successfully. • "Completed": the workflow instance completed successfully. • "Cancelled": the workflow instance was canceled. • "Failed": the execution of the workflow instance failed.
Status/Reason	Available for Status/Code "Cancelled" or "Failed".
WorkflowDefinitionCode	This is the name of the workflow definition as modeled in ION.

Element	Note
Property/NameValue	Values of the workflow input and output parameters. Must match the data type of the parameter. When included in a ProcessWorkflow with actionCode="Add", these are input parameters. When included in an AcknowledgeWorkflow with actionCode="Completed", these are output parameters.
Property/NameValue/@name	The name of the parameter as defined in the workflow model.
Property/NameValue/@type	One of the pre-defined types that you can map to the workflow parameter types. These types are supported: • IndicatorType • NumericType • IntegerNumericType • StringType • DateType • DateTimeType See the table below for an overview of mapping to workflow parameter types.
TreeProperty/TreeNode	TreeProperty contains data for a workflow structure. In the first TreeNode, these attributes are used: • ID = 1 • NodeName = the name of the structure as used in workflow • One or more NodeProperty elements with fields from the root of the structure The first node does not have a ParentID.
TreeProperty/TreeNode/ID	The unique identifier for this tree node within the document. It must only be unique within the document.
TreeProperty/TreeNode/ParentID	The ID of the node that is the parent of the current node within the tree.
TreeProperty/TreeNode/NodeName	Must be identical to the level name from the workflow structure.
TreeProperty/TreeNode/NodeProperty/NameValue	Values for the fields from the workflow structure situated on the level corresponding to the current node.
TreeProperty/TreeNode/NodeProperty/NameValue/@name	The name of the field as defined in the workflow structure.

Element	Note
TreeProperty/TreeNode/NodeProperty/NameValue/@type	The type of the field as defined in the workflow structure. These types are supported: • IndicatorType • DateType • NumericType • IntegerNumericType • StringType • DateTimeType

This table shows all available workflow parameter types and how these types are mapped to the Property types in the Workflow BOD:

Workflow Parameter Type	Property Type	Notes
Boolean	IndicatorType	Represents a true or false value. Possible values: true, false, 0, 1.
Code	StringType	The value is expected to be part of the Codes modeled in the Workflow Modeler, but no validation is enforced.
Date	DateType	The date part of a date/time stamp.
DateTime	DateTimeType	The date part and time part of a date/time stamp, separated by "T" and ending with Z (is always UTC).
Decimal	NumericType	A numeric type with a floating precision. The Decimal data type in Workflow corresponds to the double data type and is a double-precision 64-bit IEEE 754 floating point.
Hyperlink	StringType	Is displayed as a clickable link in Infor Ming.le.
Integer	IntegerNumericType	A numeric type that represents a whole number.
String	StringType	A string value of up to 4000 characters in length.

Sample workflow BODs

Sample ProcessWorkflow to start a workflow

```
<ProcessWorkflow xmlns="http://schema.infor.com/InforOAGIS/2"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLoca
tion="http://schema.infor.com/InforOAGIS/2 http://schema.infor.com/Trunk/In
forOAGIS/BODs/Developer/ProcessWorkflow.xsd"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" releaseID="11.1" version
ID="2.10.0">
   <ApplicationArea>
      <Sender>
         <LogicalID>lid://infor.test.app1</LogicalID>
         <ComponentID>ComponentID0</ComponentID>
         <TaskID>TaskID0</TaskID>
         <AuthorizationID>AuthorizationID0</AuthorizationID>
      </Sender>
      <CreationDateTime>2013-10-20T13:20:00Z</CreationDateTime>
      <BODID>infor-nid:infor:::Sample_NestedTree:1?Workflow&amp;verb=Pro
cess</BODID>
   </ApplicationArea>
   <DataArea>
      <Process>
         <TenantID>infor</TenantID>
         <ActionCriteria>
            <ActionExpression actionCode="Add"/>
         </ActionCriteria>
      </Process>
      <Workflow>
      <Status>
      <Code>Initial</Code>
      </Status>
     <WorkflowDefinitionCode>SimpleDataEntryTest</WorkflowDefinitionCode>

      <Property>
      <!-- true, false, 1 or 0 -->
      <NameValue name="aBoolean" type="IndicatorType">true</NameValue>
      </Property>
      <Property>
      <NameValue name="aCode" type="StringType">Approved</NameValue>
      </Property>
      <Property>
      <NameValue name="aDate" type="DateType">2012-12-10</NameValue>
      </Property>
      <Property>
      <NameValue name="aDateTime" type="DateTimeType">2012-12-
10T10:00:00Z</NameValue>
      </Property>
      <Property>
      <NameValue name="aDecimal" type="NumericType">453.99</NameValue>
      </Property>
```

```
      <Property>
     <NameValue name="anInteger" type="IntegerNumericType">250</NameValue>

      </Property>
      <Property>
      <NameValue name="aLink" type="StringType">http://www.inforx
treme.com</NameValue>
      </Property>
      <Property>
      <NameValue name="aString" type="StringType">This is a test
string</NameValue>
      </Property>
      <TreeProperty>
      <TreeNode>
      <ID>1</ID>
      <NodeName>Building</NodeName>
      <NodeProperty>
     <NameValue name="BuildingName" type="StringType">Office</NameValue>

      </NodeProperty>
      <NodeProperty>
     <NameValue name="NumberOfFloors" type="IntegerNumericType">1</NameVal
ue>
      </NodeProperty>
      </TreeNode>
      <TreeNode>
      <ID>2</ID>
      <ParentID>1</ParentID>
      <NodeName>Floor</NodeName>
      <NodeProperty>
      <NameValue name="FloorName" type="StringType">Ground
Floor</NameValue>
      </NodeProperty>
      </TreeNode>
      </TreeProperty>
      </Workflow>
   </DataArea>
</ProcessWorkflow>
```

Sample AcknowledgeWorkflow when the request was accepted

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<AcknowledgeWorkflow releaseID="2"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLoca
tion="http://schema.infor.com/InforOAGIS/2 http://schema.infor.com/2.5.0/In
forOAGIS/BODs/Developer/AcknowledgeWorkflow.xsd" xmlns="http://schema.in
for.com/InforOAGIS/2">
   <ApplicationArea>
      <Sender>
```

```
         <LogicalID>infor.engine.workflow</LogicalID>
         <ComponentID>ION_Workflow_Engine</ComponentID>
      </Sender>
      <CreationDateTime>2013-10-24T08:18:18.380Z</CreationDateTime>
   </ApplicationArea>
   <DataArea>
      <Acknowledge>
         <TenantID>Infor</TenantID>
         <OriginalApplicationArea>
            <Sender>
               <LogicalID>lid://infor.test.app1</LogicalID>
               <ComponentID>ComponentID0</ComponentID>
               <TaskID>TaskID0</TaskID>
               <AuthorizationID>AuthorizationID0</AuthorizationID>
            </Sender>
            <CreationDateTime>2013-10-20T13:20:00Z</CreationDateTime>
        <BODID>infor-nid:infor:::Sample_NestedTree:1?Workflow&amp;amp;verb=Pro
cess</BODID>
         </OriginalApplicationArea>
         <ResponseCriteria>
            <ResponseExpression actionCode="Accepted"/>
         </ResponseCriteria>
      </Acknowledge>
      <Workflow>
         <DocumentID>
            <ID>20</ID>
         </DocumentID>
         <Status>
            <Code>Initial</Code>
         </Status>
         <WorkflowDefinitionCode>SimpleDataEntryTest</WorkflowDefinition
Code>
      </Workflow>
   </DataArea>
</AcknowledgeWorkflow>
```

Chapter 11: Creating alerts, tasks, or notifications from an application

The Pulse engine is the component that handles alerts, tasks, and notifications as generated by Event Management and Workflow.

But any application that is connected to ION can create alerts, tasks, and notifications in Pulse directly. This is done by sending and receiving Pulse BODs such as. `PulseAlert`, `PulseTask`, `PulseNotification`.

This section explains how to create and manage alerts, tasks, and notifications through Pulse BODs.

From an application you can perform these actions:

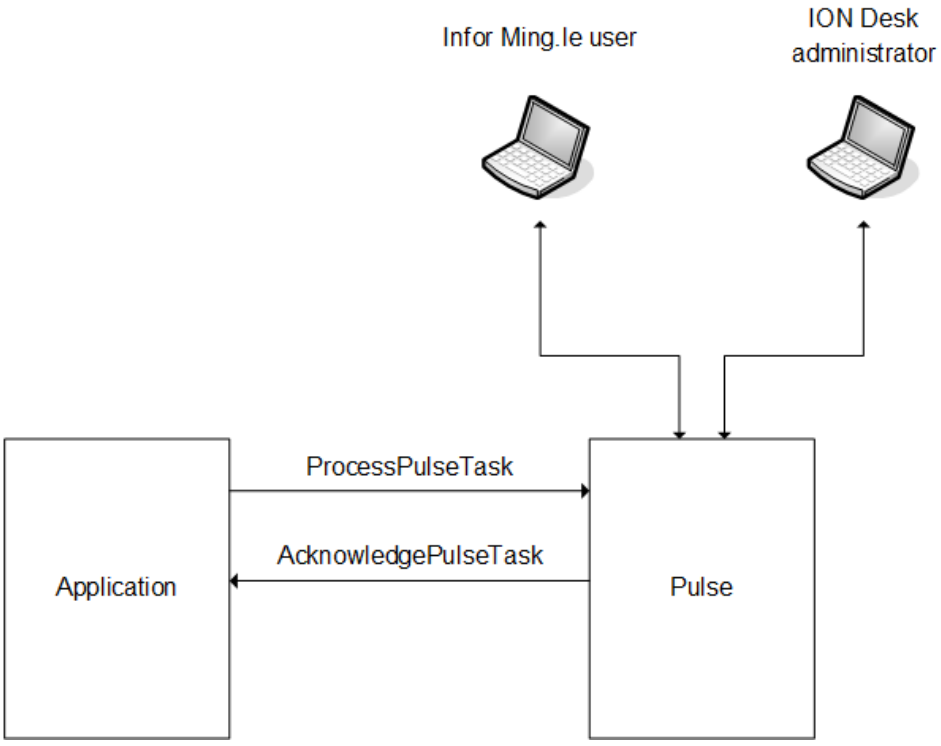
- Create alerts, tasks, or notifications.
- Receive status updates for an alert, task, or notification.
- Cancel a previously created alert, task, or notification.

Creating alerts, tasks, and notifications is not limited to applications that can send or receive BODs. For example, you can use the corresponding type of connection point to create alerts through a JMS message queue or by reading from a database.

Creating alerts, tasks, or notifications

Use `PulseTask` if the user is expected to perform a defined task. Use `PulseAlert` to notify a user of an exception. The user can decide whether to take action and then close the alert. Use `PulseNotification` to send a message 'for your information' to one or more users.

The diagram shows how tasks, alerts and notifications are created by sending Pulse BODs. In the diagram a task (`PulseTask`) is shown, but the process is the same for notifications (`PulseNotification`) and alerts (`PulseAlert`).



By publishing a `ProcessPulseTask` BOD you can create a new task in Pulse. The task is created in Pulse and an `AcknowledgePulseTask` is sent in reply.

Creating tasks from an application

To create tasks from an application:

1. Update your application so it can send `ProcessPulseTask` and to handle incoming `AcknowledgePulseTask` BODs. The `ProcessPulseTask` BOD must have action code ‘Add’ to create a new task.
2. Ensure your application is connected to ION. In the connection point, select `ProcessPulseTask` as a document to send.
3. Your application connection point must be used in at least one active document flow. If this is not the case, you can create a specific document flow containing an activity for your application and activate that flow.
4. In IFS, configure users, distribution groups, and contacts for people that must receive alerts. For users, ensure the 'Person' is filled with the value that is sent by the application as the person ID of a system user. For contacts, ensure the ‘Contact ID’ is filled with the same value that is sent by the application as the person ID of a non-system user. For distribution groups, the application must send the distribution group name as defined in IFS.

   For `PulseAlert` and `PulseNotification` BODs, the procedure is the same.

Important notes

Regarding the document flow, be aware that Pulse is an 'engine' inside ION. It is not a normal application for which you can create a connection point. Therefore, the configuration differs from a normal configuration. Normally, a document flow is created from A to B where you select the documents to be sent. This selection can be a subset of the documents sent by connection point A and received by connection point B.

If an application connection point that can publish `ProcessPulseAlert`, `ProcessPulseTask`, or `ProcessPulseNotification`, is used in an active document flow, these BODs are delivered directly to Pulse. Similarly, if the connection point is configured to receive `SyncPulseAlert`, `SyncPulseTask`, or `SyncPulseNotification`, it receives those BODs without selecting them in a document flow.

Therefore, you cannot use mapping, content-based routing, or filtering in a document flow for `PulseAlert`, `PulseTask`, and `PulseNotification`.

For details on how to define connection points and document flows, see the *Infor ION Desk User Guide*.

Creating an alert

When creating an alert, the action code (`ProcessPulseAlert/DataArea/Process/ActionCriteria/ ActionExpression/@actionCode`) must be "Add".

This table shows the elements you can use in the noun instance, `ProcessPulseAlert/DataArea/PulseAlert`:

Element	Note
Description	Required
Note	Optional
`AlertDetail` and child elements	Optional
`DistributionPerson` or `DistributionGroup`	Define at least one `DistributionPerson` or `DistributionGroup`. For a `DistributionPerson`, do not specify the `Distribution/ID`, because it is generated by Pulse.

Do not use other elements, such as DocumentID, CreationDateTime, and Status, when creating an alert. These elements are determined by the Pulse engine.

This table shows the elements that are included in the resulting AcknowledgePulseAlert BOD:

Element	Note
`DocumentID/ID`	Available if `actionCode` = "Accepted". If the alert cannot be created successfully, the `actionCode` is "Rejected".

Note: We recommend that you include a unique value for the BODID. You can use that to process the `AcknowledgePulseAlert` because the BODID is available again in the original application area.

If the alert could not be created, you receive an `AcknowledgePulseAlert` BOD with action code 'Rejected'.

If the `AcknowledgePulseAlert` BOD can not be delivered, a Confirm BOD is generated.

Creating a task

When creating a task, the action code (`ProcessPulseTask/DataArea/Process/ActionCriteria/ActionExpression/@actionCode)` must be "Add".

This table shows the elements you can use in the noun instance (`ProcessPulseTask/DataArea/PulseTask`):

Element	Note
Priority	Optional, default is MEDIUM.
Description	Required
Note	Optional
Parameter and child elements	Optional
`DistributionPerson` or `DistributionGroup`	Define at least one `DistributionPerson` or `DistributionGroup`. For a `DistributionPerson`, do not specify the `Distribution/ID`, because it is generated by Pulse.

Do not use other elements, such as DocumentID, CreationDateTime, and Status, when creating a task. These elements are determined by the Pulse engine.

This table shows the elements that are included in the resulting AcknowledgePulseTask BOD:

Element	Note
`DocumentID/ID`	Available if `actionCode` = "Accepted". If the task cannot be created successfully, the `actionCode` is "Rejected".

Note: We recommend that you include a unique value for the BODID. You can use that to process the `AcknowledgePulseTask` because the BODID is available again in the original application area.

If the task could not be created, you receive an `AcknowledgePulseTask` BOD with action code 'Rejected'.

If the `AcknowledgePulseTask` BOD can not be delivered, a Confirm BOD is generated.

Creating a notification

When creating a notification, the action code (`ProcessPulseNotification/DataArea/ Process/ActionCriteria/ActionExpression/@actionCode`) must be "Add".

This table shows the elements you can use in the noun instance (`ProcessPulseNotification/DataArea/PulseNotification`):

Element	Note
Description	Required
Parameter and child elements	Optional
`DistributionPerson` or `DistributionGroup`	Define at least one `DistributionPerson` or `DistributionGroup`. For a `DistributionPerson`, do not specify the `Distribution/ID`, because it is generated by Pulse.

Do not use other elements, such as `DocumentID`, `CreationDateTime`, and Status, when creating a notification. These elements are determined by the Pulse engine.

This table shows the elements that are included in the resulting `AcknowledgePulseNotification` BOD:

Element	Note
`DocumentID/ID`	Available if `actionCode` = "Accepted". If the notification cannot be created successfully, the `actionCode` is "Rejected".

Note: We recommend that you include a unique value for the BODID. You can use that to process the `AcknowledgePulseNotification` because the BODID is available again in the original application area.

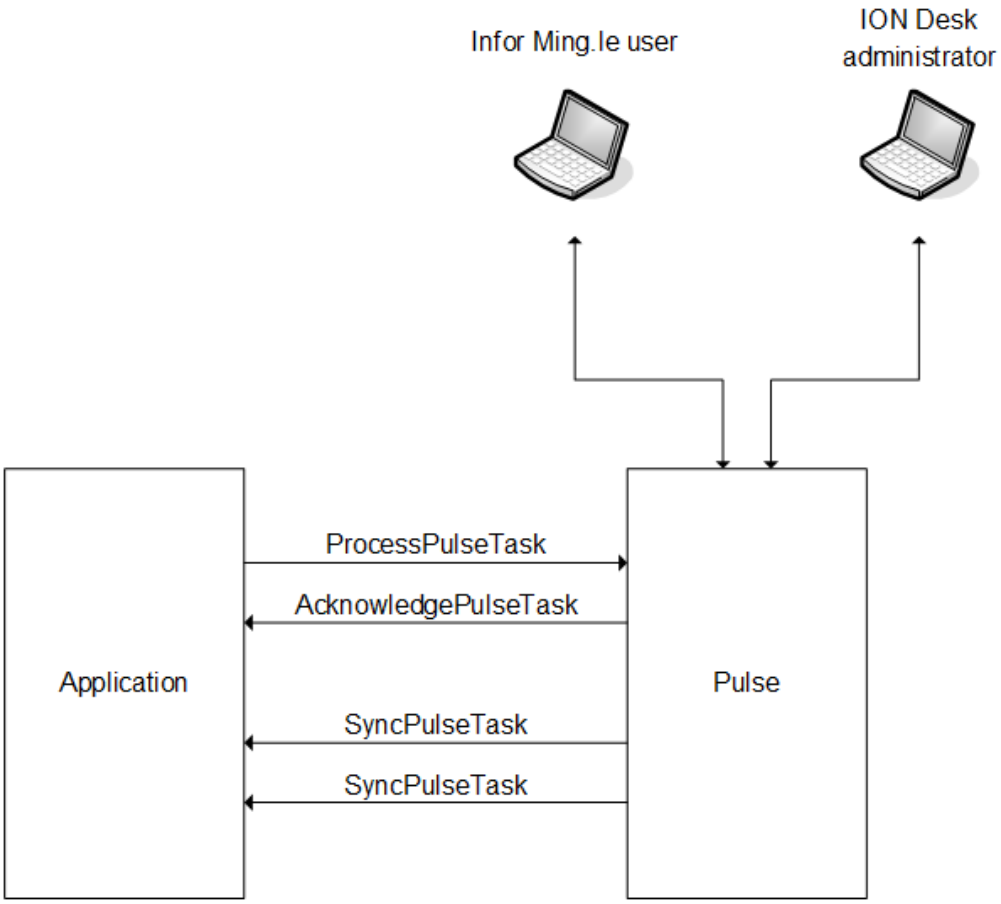
If the notification could not be created, you receive an `AcknowledgePulseNotification` BOD with action code 'Rejected'.

If the `AcknowledgePulseNotification` BOD can not be delivered, a Confirm BOD is generated.

Receiving status updates on alerts, tasks, or notifications

The Pulse engine sends Sync BODs for `PulseAlert`, `PulseTask` and `PulseNotification` to inform others about status changes in alerts, tasks and notifications.

This diagram shows a task (`PulseTask`), but the process is the same for notifications and alerts.



Pulse sends a `SyncPulseTask` when a new task is created and when the status is changed. For example, if a user picks up the task and sets it to done, a `SyncPulseTask` is sent. The application can handle this BOD to be informed about the new status. If a task is canceled through ION Desk by an administrator, a `SyncPulseTask` is also sent.

A Sync BOD is sent in these situations:

- A new task, alert or notification is created.
- An item is assigned, reassigned or unassigned.
- An item is redistributed.
- The status of an item is changed to Done or Canceled.

Pulse may not send Sync BODs for all changes to an item, such as changing a parameter value, adding a note or adding an attachment.

It is not required to handle Sync BODs when you send out Process BODs. The Acknowledge BOD tells you whether the Process request was handled successfully. When sending a `ProcessPulseNotification`, that is probably all you want to know. When creating a task, you want to know whether the task was completed and what was the outcome of the task. For example, in case of an approval task for a requisition you want to know whether the user approved or rejected the requisition. In that case you can receive the `SyncPulseTask` BODs to be informed of the task status.

It can happen that an Acknowledge Pulse BOD or a Sync Pulse BOD could be delivered to the subscribing application. In that case, a Confirm BOD is generated. The Confirm BOD contains the original Acknowledge or Sync document, and can be re-submitted later from the ION Desk UI

Receiving status updates

To receive status updates for tasks:

1. Update your application so it can handle incoming SyncPulseTask BODs. The SyncPulseAlert/DataArea/PulseTask/Source element contains the originator of the alert, so you can use this element to ignore BODs that are not relevant for your application.
2. Ensure your application is connected to ION. In the connection point, select SyncPulseTask as a document to receive.
3. Your application connection point must be used in at least one active document flow. If this is not the case, you can create a specific document flow containing an activity for your application and activate that flow.

For PulseAlert and PulseNotification BODs, the procedure is the same.

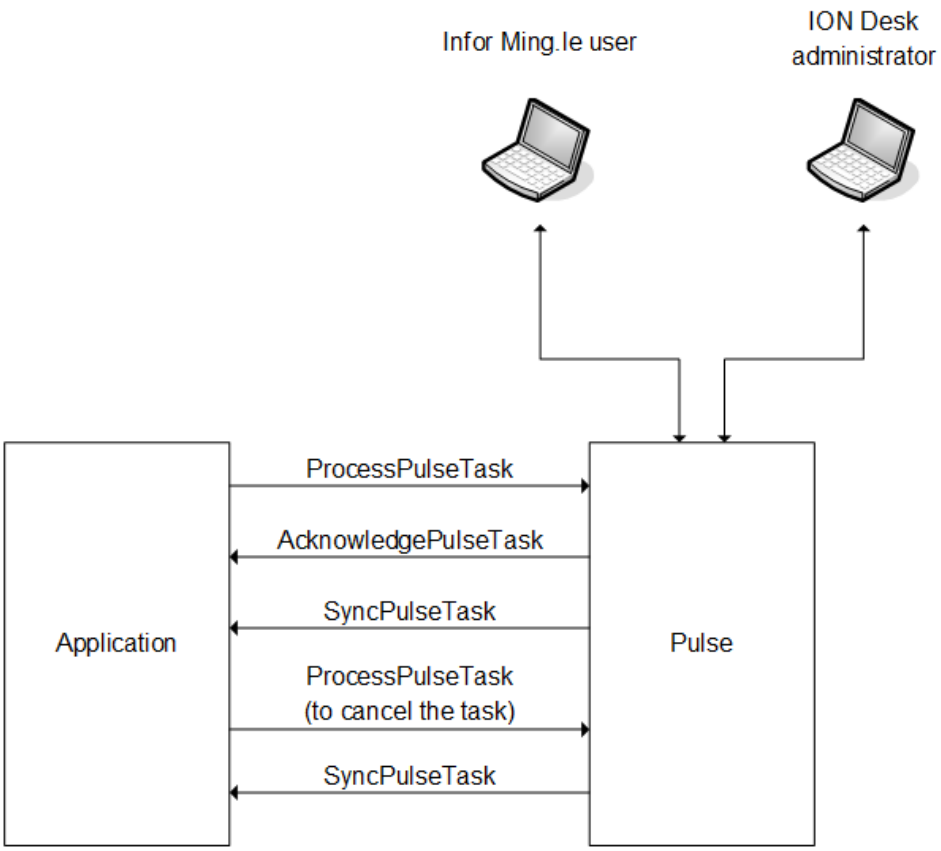
The SyncPulseAlert, SyncPulseTask, and SyncPulseNotification BODs include all elements that are relevant for the alert, task, or notification.

For the available elements, see Pulse BOD details on page 70.

Canceling alerts, tasks, or notifications

In some cases, you must cancel an alert, task, or notification before the user handled it. For example, a requisition approval task is created, but now the requestor cancels the requisition. In this case you must also cancel the task in Pulse.

This diagram shows a task (PulseTask), but the process is the same for notifications and alerts.



If you created a task by sending a ProcessPulseTask with action code 'Add', you receive an AcknowledgePulseTask which contains the ID of the task. You can use this ID to send a new ProcessPulseTask request to cancel the task.

Receiving status updates

To receive status updates for tasks:

1. Update your application so it can handle incoming SyncPulseTask BODs. The SyncPulseAlert/DataArea/PulseTask/Source element contains the originator of the alert, so you can use this element to ignore BODs that are not relevant for your application.
2. Ensure your application is connected to ION. In the connection point, select SyncPulseTask as a document to receive.
3. Your application connection point must be used in at least one active document flow. If this is not the case, you can create a specific document flow containing an activity for your application and activate that flow.

For PulseAlert and PulseNotification BODs, the procedure is the same.

The SyncPulseAlert, SyncPulseTask, and SyncPulseNotification BODs include all elements that are relevant for the alert, task, or notification.

For the available elements, see Pulse BOD details on page 70.

Canceling an alert

When canceling an alert, the action code (ProcessPulseAlert/DataArea/Process/ActionCriteria/ActionExpression/@actionCode) must be "Change".

This table shows the elements you must use in the noun instance (ProcessPulseAlert/DataArea/PulseAlert):

Element	Note
DocumentID/ID	Use the DocumentID/ID as provided in the Acknowledge BOD when the item was created.
Status/Code	Use value 'CANCELLED'.

When processing the cancel request in the Pulse engine, an Acknowledge BOD is sent in reply. If the item was canceled successfully, the actionCode of the BOD is "Accepted". If the item could not be canceled, the actionCode is "Rejected".

You can only cancel an alert when:

- The alert is open. If the user completed the alert, you can no longer cancel it.
- The alert is created by a ProcessPulseAlert BOD. If the alert is created by a Monitor you cannot cancel the alert using a ProcessPulseAlert BOD.

Canceling a task

When canceling a task, the action code (ProcessPulseTask/DataArea/Process/ActionCriteria/ActionExpression/@actionCode) must be "Change".

This table shows the elements you must use in the noun instance (ProcessPulsetask/DataArea/PulseTask):

Element	Note
DocumentID/ID	Use the DocumentID/ID as provided in the Acknowledge BOD when the item was created.
Status/Code	Use value 'CANCELLED'.

When processing the cancel request in the Pulse engine, an Acknowledge BOD is sent in reply. If the item was canceled successfully, the actionCode of the BOD is "Accepted". If the item could not be canceled, the actionCode is "Rejected".

You can only cancel a task when:

- The task is open. If the user completed the task, you can no longer cancel it.
- The task is created by a ProcessPulseTask BOD. If the task is created by a Workflow you cannot cancel the task using a ProcessPulseTask BOD.

Canceling a notification

When canceling a notification, the action code (ProcessPulseNotification/DataArea/Process/ActionCriteria/ActionExpression/@actionCode) must be "Change".

This table shows the elements you must use in the noun instance (ProcessPulseNotification/DataArea/PulseNotification):

Element	Note
DocumentID/ID	Use the DocumentID/ID as provided in the Acknowledge BOD when the item was created.
Status/Code	Use value 'CANCELLED'.

When processing the cancel request in the Pulse engine, an Acknowledge BOD is sent in reply. If the item was canceled successfully, the actionCode of the BOD is "Accepted". If the item could not be canceled, the actionCode is "Rejected".

You can only cancel a notification when:

- The notification is open. If the user completed the task, you can no longer cancel it.
- The notification is created by a ProcessPulseNotification BOD. If the notification is created by a Workflow you cannot cancel the notification using a ProcessPulseNotification BOD.

Pulse BOD details

The definition of ProcessPulseAlert, AcknowledgePulseAlert, and SyncPulseAlert is available on this site:

http://schema.infor.com

This section contains an explanation on the elements that exist in these documents.

PulseAlert

This table shows the elements in PulseAlert:

Element	Note
DocumentID/ID	Unique identification of the alert.
CreationDateTime	Date when the alert was created in Pulse.
LastModificationDateTime	Date when the alert was last modified.

Element	Note
Status/Code	Status of the alert. Values are: • NEW - The initial status for a new item. • ASSIGNED - Assigned to a specific user. • UNASSIGNED - No longer assigned to a specific user. • DONE - Completed. • CANCELLED - canceled by an administrator (through ION Desk) or by the application (through a ProcessPulseAlert BOD).
IsEscalated	This indicator has value true if an alert is escalated and false otherwise. This element is only supported in Sync messages.
EscalationLevel	The number of levels in the organizational hierarchy to which the alert is escalated. If the alert is not escalated the value is 0. Otherwise the EscalationLevel is greater than 0.
DueDateTime	Date and time when the alert is due. This element is shown only if a due date was configured in the monitor that created this alert. This element is supported only in the Sync messages.
Description	The description that is displayed to the end user as the summary of the alert. You can use hash tags to make searching easier. For example: `Late shipment for #sales order 25 of customer #acme` To define a category, use `##` at the end of the message. To include values from the alert details, use square brackets around the parameter labels. To use the actual characters for square brackets, use an escape character: `\[` or `\]`
Description/@languageID	The language code of the Description field. For details, see the notes in the "Supported features" section.
Note	Notes that are added by people who handled the alert. Notes can be added, but cannot be modified or removed.
Note/@userID	The ID (personId) of the person who added the note.

Element	Note
Note/@author	Full name of the person who added the note. This attribute is required when a Note is added through a ProcessPulseAlert BOD." / "ProcessPulseTask BOD." / "ProcessPulseNotification BOD.
Note/@entryDateTime	The date/time at which the note was created.
Note/@noteID	Identification of the note within the alert.
Source/Type	BOD: the alert was created by sending a ProcessPulseAlert BOD. MONITOR: the alert was created by an event monitor.
Source/Name	If Type=BOD: the logical ID of the sender of the ProcessPulseAlert BOD. For example, `lid://infor.erp.myerp` If Type is MONITOR: the name of the monitor that created the alert. For example, `MyMonitor`.
AlertDetail	Details of the alert that are displayed to the user. Each AlertDetail group contains document references or trees. For example, an AlertDetail can contain one or more document references followed by a tree containing data for an order and its order lines.
AlertDetail/@sequence	The sequence number of the AlertDetail group. This is used for ordering the alert details when displaying them to a user.

Element	Note
AlertDetail/PulseDocumentReference	Reference to another business document. The format is the same as the standard DocumentReference, but additionally it has a sequence attribute. For example: `<PulseDocumentReference> type="SalesOrder" sequence="1"> <DocumentID> <ID accountingEntity="infor" location="bvld" lid="lid://infor.ln.440"> ORD0015236</ID> <RevisionID>123</RevisionID> </DocumentID> </PulseDocumentReference>` The RevisionID tag is only used if the document referred from this alert contains a RevisionID.
AlertDetail/PulseDocumentReference/@sequence	Sequence number to indicate the sequence in which the document references must be displayed to the user when showing the alert details.
AlertDetail/TreeNode	Node in the data tree. Alert data is a tree structure to enable multi-level data objects, such as an order header having order lines.
AlertDetail/TreeNode/@sequence	If the tree node is a child of another node, the sequence attribute defines the sequence of the child nodes relative to their parent.
AlertDetail/TreeNode/ID	Identification for this tree node within the alert.
AlertDetail/TreeNode/ParentID	Omitted if the tree node is the top-level node in the tree. Otherwise it contains the ID of the parent node. The parent node must exist within the same AlertDetail. In an AlertDetail, all TreeNodes except one will have a ParentID.
AlertDetail/TreeNode/NodeName	The name of the node. This name is displayed to the user. For example: `Sales Order`.

Element	Note
AlertDetail/TreeNode/TreeNodeParameter	A parameter in a tree node, which contains a data element that can be displayed to the user who handles the alert. Alert parameters are always read-only. For example: `<TreeNodeParameter sequence="1"> <Name>OrderNumber</Name> <Value>12345</Value> <DataType listID="Pulse Datatypes">STRING </DataType> <Label>Order Number</Label> </TreeNodeParameter>`
AlertDetail/TreeNode/TreeNodeParameter/Name	The name that identifies the parameter within the TreeNode. This element is required for each parameter.
AlertDetail/TreeNode/TreeNodeParameter/Value	The (serialized) value of the parameter. The formatting depends on the DataType and is the same as the formatting that is normally used in BOD data elements.
AlertDetail/TreeNode/TreeNodeParameter/DataType	The data type of the parameter. This element is required for each parameter. You can use these data types: • STRING - a string value that is up to 4000 characters in length • INTEGER - a numeric type that represents a whole number • DECIMAL - a numeric type that has a floating precision. Values may be expressed using the scientific e-notation. For details about the scientific e-notation, see Wikipedia or other resources on the internet. • BOOLEAN - represents a true or false value • DATETIME - the date part and time part of a date/time stamp separated by "T" and ending with Z (is always UTC) • TIME - the time part of a date/time stamp • DATE - the date part of a date/time stamp • DURATION - time interval, starting with P followed by nM (minutes) or nH (hours) or nD (days). For example, P2D3H.

Element	Note
AlertDetail/TreeNode/TreeNodeParameter/Label	The label of the parameter that is used when displaying the parameter to a user. This element is required for each parameter.
AssignedPerson	The user to which the alert is currently assigned.
AssignedPerson/PersonReference	Reference to a person.
AssignedPerson/PersonReference/IDs/ID	Identifier of the person. The ID must be a Person ID or Contact ID as defined in IFS (case-sensitive).
AssignedPerson/PersonReference/Name	Name of the person.
AssignedPerson/PersonReference/SystemUserIndicator	Value is `true` if the person is a User in IFS. Value is `false` if the person is a Contact in IFS.
DistributionPerson	Person in the distribution list for the alert. If the alert is not assigned to a person, one of these persons can pick it up.
DistributionPerson/ID	Identification of the distribution person within the alert
DistributionPerson/PersonReference	Reference to a person.
DistributionPerson/PersonReference/IDs/ID	Identifier of the person. The ID must be a Person ID or Contact ID as defined in IFS (case-sensitive). This element is required for each distribution person.
DistributionPerson/PersonReference/Name	Name of the person.
DistributionPerson/PersonReference/SystemUserIndicator	Value is `true` if the person is a User in IFS. Value is `false` if the person is a Contact in IFS. This element is required for each distribution person.
DistributionGroup	Specify one or more distribution groups to which the alert must be distributed. This element can be used in addition to the DistributionPerson element, or instead of the DistributionPerson element.
DistributionGroup/Name	Identifier of the distribution group as defined in Infor Ming.le User Management. The alert is distributed to all users that are members of this group at the time the alert is created.
DistributionGroup/Description	Description of the distribution group. This element is optional and is not used to determine the distribution list.

Element	Note
DistributionGroup/Description/@languageID	Describe the language code of the description element. This attribute is optional and is not used for the distribution functionality.

Note: The DistributionGroup element is supported only in the ProcessPulseAlert BOD. A SyncPulseAlert BOD can be sent after the creation of the alert. In that case, the distribution list of the alert is described using a DistributionPerson element for each user from the distribution group.

PulseTask

This table shows the elements in PulseTask:

Element	Note
DocumentID/ID	Unique identification of the task.
CreationDateTime	Date when the task was created in Pulse.
LastModificationDateTime	Date when the task was last modified.
Status/Code	Status of the task. Values are: • NEW - The initial status for a new item. • ASSIGNED - Assigned to a specific user. • UNASSIGNED - No longer assigned to a specific user. • DONE - Completed. • CANCELLED - canceled by an administrator (through ION Desk) or by the application (through a ProcessPulseTask BOD).
IsEscalated	This indicator has value true if a task is escalated and false otherwise. This element is only supported in Sync messages.
EscalationLevel	The number of levels in the organizational hierarchy to which the alert is escalated. If the task is not escalated the value is 0. Otherwise the EscalationLevel is greater than 0.
Priority	The priority of the task. Values are: • HIGH • MEDIUM • LOW
DueDateTime	Date and time when the task is due. This element is shown only if a due date was configured in the workflow task properties. This element is supported only in the Sync messages.

Element	Note
Description	The description that is displayed to the end user as the summary of the task. You can use hash tags to make searching easier. For example: `Approve #requisition 25` To define a category, use `##` at the end of the message. To include values from the alert details, use square brackets around the parameter labels. To use the actual characters for square brackets, use an escape character: `\[` or `\]`
Description/@languageID	The language code of the Description field. For details, see the notes in the "Supported features" section.
Note	Notes that are added by people who handled the alert. Notes can be added, but cannot be modified or removed.
Note/@userID	The ID (personId) of the person who added the note.
Note/@author	Full name of the person who added the note. This attribute is required when a Note is added through a ProcessPulseAlert BOD." / "ProcessPulseTask BOD." / "ProcessPulseNotification BOD.
Note/@entryDateTime	The date/time at which the note was created.
Note/@noteID	Identification of the note within the task.
Note/@type	If this attribute is missing or empty "", this note is a current note of this Task. If the @type attribute is filled with a string other than "", this note is a propagated Note, which was specified in a previous Task of the same workflow.
Source/Type	BOD: the task was created by sending a ProcessPulseTask BOD. WORKFLOW: the task was created by a workflow.

Element	Note
Source/Name	If Type=BOD: the logical ID of the sender of the ProcessPulseTask BOD. For example, `lid://infor.erp.myerp` If Type is WORKFLOW: the name of the workflow definition that created the task. For example, `My Workflow`.
Parameter	Data of the task that is displayed to the user and optionally can be updated by the user. For example: `<Parameter sequence="1"> <Name>OrderNumber</Name> <Value>12345</Value> <DataType listID="Pulse Datatypes">STRING</DataType> <Label>Order Number</Label> <ReadOnlyIndicator>true</ReadOnlyIndicator> </Parameter>`
Parameter/@sequence	The sequence number of the parameter. This is used for ordering the parameters when displaying them to a user. You are only allowed to use this attribute if there are no elements of type TreeParameter in the document.
Parameter/Sequence	You must specify this element in combination with TreeParameter/Sequence. This element is used to determine the order in which Parameters and TreeParameters are displayed to the user.
Parameter/Name	The name that identifies the parameter within the task. This element is required for each parameter.
Parameter/Value	The (serialized) value of the parameter. The formatting depends on the DataType and is the same as the formatting that is normally used in BOD data elements.

Element	Note
Parameter/DataType	The data type of the parameter. This element is required for each parameter. You can use these data types: • STRING - a string value that is up to 4000 characters in length • INTEGER - a numeric type that represents a whole number • DECIMAL - a numeric type that has a floating precision. Values may be expressed using the scientific e-notation. For details about the scientific e-notation, see Wikipedia or other resources on the internet. • BOOLEAN - represents a true or false value • DATETIME - the date part and time part of a date/time stamp separated by "T" and ending with Z (is always UTC) • TIME - the time part of a date/time stamp • DATE - the date part of a date/time stamp
Parameter/Label	The label of the parameter that is used when displaying the parameter to a user. This element is required for each parameter.
Parameter/Label /@languageID	The language code of the Label field. For details, see the notes in the "Supported features" section.
Parameter/ReadyOnlyIndicator	Indicates whether the parameter is read-only: • If the value is `true`, the user is not allowed to change the value. • If the value is `false`, the user can change the value when handling the item. This element is required for each parameter.
Parameter/Restriction	Restriction to the data type. This element can contain the name of a code as defined in ION Desk. The values that the user can specify are then restricted to the specified code.
TreeParameter	The TreeParameter consists of a TreeDefinition and TreeNodes specified after the TreeDefinition. The TreeDefinition defines a complex data structure. The TreeNodes define the data value for the structure. The TreeDefinition and the TreeNodes must be consistent.

Element	Note
TreeParameter/Sequence	Indicates the sorting sequence in which TreeParameters and Parameters must be displayed to the user in the task. If one Sequence is defined, all Parameters must specify a Sequence element.
TreeParameter/TreeDefinition	Definition for the data structure contained in the TreeParameter. Definition includes several unique TreeNodes with their ID, ParentID, NodeName, and TreeNodeParameters.
TreeParameter/TreeDefinition/TreeNode	A node in a tree using a parent relationship and containing the definition of the properties required for the node.
TreeParameter/TreeDefinition/TreeNode/Sequence	Indicates the sequence of the node within the TreeParameter. You must specify the Sequence for all TreeNodes if this element is specified for the root TreeNode. The ordering of the TreeNodes within the structure is based on the specified Sequence. If the Sequence is not specified for the root TreeNode, the structure is sorted based on the xml.
TreeParameter/TreeDefinition/TreeNode/ID	The unique identifier for this tree node within this definition of a TreeParameter.
TreeParameter/TreeDefinition/TreeNode/ParentID	The ID of the node that is the parent of this node within the tree. The root TreeNode does not have a ParentID.
TreeParameter/TreeDefinition/TreeNode/NodeName	This is the unique identification of a tree node.
TreeParameter/TreeDefinition/TreeNode/Label	The label of the parameter that is used when showing the parameter to a user. You must specify at least one label. You can specify several labels, each with a different languageID for translated labels.
TreeParameter/TreeDefinition/TreeNode/Label/@languageID	The language code of the Label field. For details, see notes from Supported Features.
TreeParameter/TreeDefinition/TreeNode/TreeNodeParameter	The TreeNodeParameter defines a property within a TreeNode level. TreeNodeParameters are optional. Values for these properties are specified in the TreeNodes that follow the TreeDefinition.
TreeParameter/TreeDefinition/TreeNode/TreeNodeParameter/Sequence	Indicates the sequence of the TreeNodeParameter within the TreeNode.
TreeParameter/TreeDefinition/TreeNode/TreeNodeParameter/Name	The name of the TreeNodeParameter is used to identify this property in the tree node instances to specify its value.

Element	Note
TreeParameter/TreeDefinition/TreeNode/TreeNodeParameter/DataType	The data type of the associated value. You can use these data types: • STRING - a string value that is up to 4000 characters in length • INTEGER - a numeric type that represents a whole number • DECIMAL - a numeric type that has a floating precision. Values may be expressed using the scientific e-notation. For details about the scientific e-notation, see Wikipedia or other resources on the internet. • BOOLEAN - represents a true or false value • DATETIME - the date part and time part of a date/time stamp separated by "T" and ending with Z (is always UTC) • TIME - the time part of a date/time stamp • DATE - the date part of a date/time stamp
TreeParameter/TreeDefinition/TreeNode/TreeNodeParameter/Label	The label of the node that is used when displaying it to a user. You must specify at least one label. You can specify several labels, each with a different languageID for translated labels.
TreeParameter/TreeDefinition/TreeNode/TreeNodeParameter/Label/@languageID	The language code of the Label field. For details, see notes from Supported Features.
TreeParameter/TreeNode	A node in a tree using a parent relationship and containing properties required for the node and their values.
TreeParameter/TreeNode/ID	The unique identifier for this tree node within this TreeParameter.
TreeParameter/TreeNode/ParentID	The ID of the node that is the parent of this node within the tree definition.
TreeParameter/TreeNode/NodeName	The unique identification of a tree node. This name must match a node name from the TreeDefinition. Several TreeNode instances with the same NodeName can exist.
TreeParameter/TreeNode/TreeNodeParameter	List of properties for this TreeNode.
TreeParameter/TreeNode/TreeNodeParameter/Name	The property name that must match with a property defined for this TreeNode in the TreeDefinition.
TreeParameter/TreeNode/TreeNodeParameter/Value	The value for this tree node property. The value must be consistent with the data type specified in the tree definition.

Element	Note

ActionParameter	Parameter that holds the actions that a user can do when closing the task.
	For example:
	<pre><ActionParameter>
	<Name>ApprovalResult</Name>
	<Value>Rejected</Value>
	<Action sequence="1">
	<Value>Approved</Value>
	<Label>Approve</Label>
	</Action>
	<Action sequence="2">
	<Value>Rejected</Value>
	<Label>Reject</Label>
	</Action>
	</ActionParameter></pre>
	An action parameter must have at least one action.

ActionParameter/Name	The name that identifies the action parameter within the task. This element is required for each action parameter.
ActionParameter/Value	The (serialized) value of the action parameter.
ActionParameter/Action/Value	The value that is assigned to the action parameter when the user selects this action.
ActionParameter/Action/Label	The label used when displaying the action button to the user. This element is required for each action parameter. Even though it is technically possible to use a string of maximum length of 255 characters, we recommend that you use short labels that are suitable for action buttons.
ActionParameter/Action/Label /@languageID	The language code of the Label field. For details, see notes from Supported Features.
AssignedPerson	The user to which the notification is currently assigned. A notification is initially distributed to all users having a DistributionPerson defined in the notification.
AssignedPerson/PersonReference	Reference to a person.
AssignedPerson/PersonReference/IDs/ID	Identifier of the person. The ID must be a Person ID or Contact ID as defined in IFS (case-sensitive).
AssignedPerson/PersonReference/Name	Name of the person.
AssignedPerson/PersonReference/SystemUserIndicator	Value is `true` if the person is a User in IFS. Value is `false` if the person is a Contact in IFS.

Element	Note
DistributionPerson	Person in the distribution list for the task. The task is distributed to all users for which a DistributionPerson is included.
DistributionPerson/ID	Identification of the distribution person within the distribution list of the task.
DistributionPerson/PersonReference	Reference to a person.
DistributionPerson/PersonReference/IDs/ID	Identifier of the person. The ID must be a Person ID or Contact ID as defined in IFS (case-sensitive). This element is required for each distribution person.
DistributionPerson/PersonReference/Name	Name of the person.
DistributionPerson/PersonReference/SystemUserIndicator	Value is `true` if the person is a User in IFS. Value is `false` if the person is a Contact in IFS. This element is required for each distribution person. For PulseTask, the value must always be `true`.
DistributionGroup	Specify one or more distribution groups to which the task must be distributed. This element can be used in addition to the DistributionPerson element, or instead of the DistributionPerson element.
DistributionGroup/Name	Identifier of the distribution group as defined in Infor Ming.le User Management. The task is distributed to all users that are members of this group at the time the task is created.
DistributionGroup/Description	Description of the distribution group. This element is optional and is not used to determine the distribution list.
DistributionGroup/Description/@languageID	Describe the language code of the description element. This attribute is optional and is not used for the distribution functionality.

Note: The DistributionGroup element is supported only in the ProcessPulseTask BOD. A SyncPulseTask BOD can be sent after the creation of the task. In that case, the distribution list of the task is described using a DistributionPerson element for each user from the distribution group.

PulseNotification

This table shows the elements in PulseNotification:

Element	Note
DocumentID/ID	Unique identification of the notification.
CreationDateTime	Date when the notification was created in Pulse.
LastModificationDateTime	Date when the notification was last modified.
Status/Code	Status of the notification. Values are: • ASSIGNED - The initial status for a new item. The notification is assigned to each of the distribution persons. • DONE - Completed. • CANCELLED - canceled by an administrator (through ION Desk) or by the application (through a ProcessPulseNotification BOD).
Description	The description that is displayed to the end user as the summary of the notification. You can use hash tags to make searching easier. For example: `#Requisition 25 is approved` To define a category, use `##` at the end of the message. To include values from the alert details, use square brackets around the parameter labels. To use the actual characters for square brackets, use an escape character: `\[` or `\]`
Description/@languageID	The language code of the Description field. For details, see the notes in the "Supported features" section.
Note	Propagated notes that are added by users who worked on Tasks from the same workflow. You cannot add or remove notes from a Notification. This field is only applicable for Sync.PulseNotification for notifications created by Workflow.
Note/@userID	The ID (personId) of the person who added the note.
Note/@author	Full name of the person who added the note. This attribute is required when a Note is added through a ProcessPulseAlert BOD." / "ProcessPulseTask BOD." / "ProcessPulseNotification BOD.
Note/@entryDateTime	The date/time at which the note was created.
Note/@noteID	Identification of the note within the notification.

Element	Note
Source/Type	BOD: the notification was created by sending a ProcessPulseNotification BOD. WORKFLOW: the notification was created by a workflow.
Source/Name	If Type=BOD: the logical ID of the sender of the ProcessPulseNotification BOD. For example, `lid://infor.erp.myerp` If Type is WORKFLOW: the name of the workflow definition that created the notification. For example, `MyWorkflow`.
Parameter	Data of the notification that is displayed to the user. Notification parameters are always read-only. For example: `<Parameter sequence="1">   <Name>OrderNumber</Name>   <Value>12345</Value>   <DataType   listID="Pulse Datatypes">STRING</DataType>   <Label>Order Number</Label> </Parameter>`
Parameter/@sequence	The sequence number of the parameter. This is used for ordering the parameters when displaying them to a user. It is only allowed to use this attribute if there are no elements of type TreeParameter in the document.
Parameter/Sequence	You must specify this element in combination with TreeParameter/Sequence. This element is used to determine the order in which Parameters and TreeParameters are displayed to the user.
Parameter/Name	The name that identifies the parameter within the notification. This element is required for each parameter.
Parameter/Value	The (serialized) value of the parameter. The formatting depends on the DataType and is the same as the formatting that is normally used in BOD data elements.

Element	Note
Parameter/DataType	The data type of the parameter. This element is required for each parameter. You can use these data types: • STRING - a string value that is up to 4000 characters in length • INTEGER - a numeric type that represents a whole number • DECIMAL - a numeric type that has a floating precision. Values may be expressed using the scientific e-notation. For details about the scientific e-notation, see Wikipedia or other resources on the internet. • BOOLEAN - represents a true or false value • DATETIME - the date part and time part of a date/time stamp separated by "T" and ending with Z (is always UTC) • TIME - the time part of a date/time stamp • DATE - the date part of a date/time stamp
Parameter/Label	The label of the parameter that is used when displaying the parameter to a user. This element is required for each parameter.
Parameter/Label /@languageID	The language code of the Label field. For details, see the notes in the "Supported features" section.
TreeParameter	The TreeParameter consists of a TreeDefinition and TreeNodes specified after the TreeDefinition. The TreeDefinition defines a complex data structure. The TreeNodes specify the data value for the structure. The TreeDefinition and the TreeNodes must be consistent.
TreeParameter/Sequence	Indicates the sorting sequence in which TreeParameters and Parameters must be displayed to the user in the notification. If one Sequence is specified, all the Parameters must specify a Sequence element.
TreeParameter/TreeDefinition	Definition for the data structure contained in the TreeParameter. This definition includes several unique TreeNodes with their ID, ParentID, NodeName, and TreeNodeParameters.
TreeParameter/TreeDefinition/TreeNode	A node in a tree using a parent relationship and containing the definition of the properties required for the node.

Element	Note
TreeParameter/TreeDefinition/TreeNode/Sequence	Indicates the sequence of the node within the TreeParameter. The Sequence must be specified for all the TreeNodes if this element is specified for the root TreeNode. The ordering of the TreeNodes within the structure is based on the specified Sequence. If the Sequence is not specified for the root TreeNode, the structure is sorted based on the xml.
TreeParameter/TreeDefinition/TreeNode/ID	The unique identifier for this tree node within this definition of a TreeParameter.
TreeParameter/TreeDefinition/TreeNode/ParentID	The ID of the node that is the parent of this node within the tree. The root TreeNode does not have a ParentID.
TreeParameter/TreeDefinition/TreeNode/NodeName	This is the unique identification of a tree node.
TreeParameter/TreeDefinition/TreeNode/Label	The label of the parameter that is used when displaying the parameter to a user. You must specify at least one label. You can specify several labels, each with a different languageID for translated labels.
TreeParameter/TreeDefinition/TreeNode/Label/@languageID	The language code of the Label field. For details, see the notes in the "Supported features" section.
TreeParameter/TreeDefinition/TreeNode/TreeNodeParameter	Defines a property within a TreeNode level. TreeNodeParameters are optional. Values for these properties are specified in the TreeNodes that follow the TreeDefinition.
TreeParameter/TreeDefinition/TreeNode/TreeNodeParameter/Sequence	Indicates the sequence of the TreeNodeParameter within the TreeNode.
TreeParameter/TreeDefinition/TreeNode/TreeNodeParameter/Name	The name of the TreeNodeParameter is used to identify this property in the tree node instances to specify its value.

Element	Note
TreeParameter/TreeDefinition/TreeNode/TreeNodeParameter/DataType	The data type of the associated value. You can use these data types: • STRING - a string value that is up to 4000 characters in length • INTEGER - a numeric type that represents a whole number • DECIMAL - a numeric type that has a floating precision. Values may be expressed using the scientific e-notation. For details about the scientific e-notation, see Wikipedia or other resources on the internet. • BOOLEAN - represents a true or false value • DATETIME - the date part and time part of a date/time stamp separated by "T" and ending with Z (is always UTC) • TIME - the time part of a date/time stamp • DATE - the date part of a date/time stamp
TreeParameter/TreeDefinition/TreeNode/TreeNodeParameter/Label	The label of the node that is used when it is displayed to a user. You must specify at least one label. You can specify several labels, each with a different languageID for translated labels.
TreeParameter/TreeDefinition/TreeNode/TreeNodeParameter/Label/@languageID	The language code of the Label field. For details, see the notes in the "Supported features" section.
TreeParameter/TreeNode	A node in a tree using a parent relationship and containing properties required for the node and their values.
TreeParameter/TreeNode/ID	The unique identifier for this tree node within this TreeParameter.
TreeParameter/TreeNode/ParentID	The ID of the node that is the parent of this node within the tree definition.
TreeParameter/TreeNode/NodeName	The unique identification of a tree node. This name must match a node name from the TreeDefinition. Several TreeNode instances with the same NodeName can exist.
TreeParameter/TreeNode/TreeNodeParameter	List of properties for this TreeNode.
TreeParameter/TreeNode/TreeNodeParameter/Name	The property name that must match with a property specified for this TreeNode in the TreeDefinition.
TreeParameter/TreeNode/TreeNodeParameter/Value	The value for this tree node property. The value must be consistent with the data type defined in the tree definition.

Element	Note
AssignedPerson	The user to which the notification is currently assigned. When a user closes the notification, he or she is removed from the list of assigned persons.
AssignedPerson/PersonReference	Reference to a person.
AssignedPerson/PersonReference/IDs/ID	Identifier of the person. The ID must be a Person ID or Contact ID as defined in IFS (case-sensitive).
AssignedPerson/PersonReference/Name	Name of the person.
AssignedPerson/PersonReference/SystemUserIndicator	Value is `true` if the person is a User in IFS. Value is `false` if the person is a Contact in IFS.
DistributionPerson	Person in the distribution list for the notification. The notification is send to each of the distribution persons in parallel.
DistributionPerson/ID	Identification of the distribution person within the notification.
DistributionPerson/PersonReference	Reference to a person.
DistributionPerson/PersonReference/IDs/ID	Identifier of the person. The ID must be a Person ID or Contact ID as defined in IFS (case-sensitive). This element is required for each distribution person.
DistributionPerson/PersonReference/Name	Name of the person.
DistributionPerson/PersonReference/SystemUserIndicator	Value is `true` if the person is a User in IFS. Value is `false` if the person is a Contact in IFS. This element is required for each distribution person.
DistributionGroup	Specify one or more distribution groups to which the notification must be distributed. This element can be used in addition to the DistributionPerson element, or instead of the DistributionPerson element.
DistributionGroup/Name	Identifier of the distribution group as defined in Infor Ming.le User Management. The notification is distributed to all users that are members of this group at the time the notification is created.
DistributionGroup/Description	Description of the distribution group. This element is optional and is not used to determine the distribution list.
DistributionGroup/Description/@languageID	Describe the language code of the description element. This attribute is optional and is not used for the distribution functionality.

Note: The DistributionGroup element is supported only in the ProcessPulseNotification BOD. A SyncPulseNotification BOD can be sent after the creation of the notification. In that case, the distribution list of the notification is described using a DistributionPerson element for each user from the distribution group.

Supported features

Most features of Pulse are supported when creating an alert, task, or notification through a process BOD. This table shows which features of Pulse are supported when creating an alert, task, or notification through a Process BOD.

Feature	Process PulseAlert	Process PulseTask	Process PulseNotification
Message (Description)	Yes	Yes	Yes
Priority	N/A	Yes	N/A
Defined sequence of data elements	Yes	Yes	Yes
Hierarchy of data	Yes	Yes	Yes
Data labels	Equal to names	Yes	Yes
Read-only values	Yes	Yes	Yes
Editable values	N/A	Yes	N/A
Data-type dependent controls	N/A	Yes	N/A
Action buttons	N/A	Yes	N/A
Distribution to (Pulse) users	Yes	Yes	Yes
Distribution to external contacts (e-mail)	Yes	N/A	Yes
Distribution to groups	Yes	Yes	Yes
Escalation policies	No	No	N/A
Translations	No See note 1.	No See note 1.	No See note 1.
Locales	Yes See note 2.	Yes See note 2.	Yes See note 2.
Hyperlinks	N/A	Yes See note 3.	Yes See note 3.

Feature	Process PulseAlert	Process PulseTask	Process PulseNotification
Document references with drill-back	Yes	N/A	N/A
Attach notes	Yes	Yes	N/A
Add attachments	Yes See note 4.	Yes See note 4.	Yes See note 4.
Start workflow from alert	No	N/A	N/A

Notes:

1 Translations are supported for Description and Label fields, in combination with the languageID attribute. An occurrence of a Description or Label field that does not have the languageID attribute is called the 'default value' and is required. Optionally, you can add additional occurrences of these fields, called 'translations', each having a languageID attribute filled with a language. At runtime, Infor Ming.le matches the display language from the user's regional settings with the language code from the translations and shows the corresponding string. If there is no match, the default value is displayed.

Language codes are the codes used by Microsoft.

See also: ISO 639 http://en.wikipedia.org/wiki/ISO_639

2 Parameter values are displayed in Infor Ming.le using the user's local settings, such as date and time format, time zone, and number format. This depends on the parameter's data type. Data in the Description element is not formatted by ION, but displayed as is. For example when the Description contains: "On 24-12-2012T14:50:40Z the total amount was 1,234.56".

3 For tasks and notifications, you can use a hyperlink as a parameter value. In that case, use data type STRING. A string is presented as a hyperlink to the Infor Ming.le user if it starts with 'http://' or 'https://'.

4 You can use attachments in Infor Ming.le widgets, also for items that are created using a Process BOD. But you cannot include them in a BOD. For tasks and notifications, you can include hyperlinks in the BOD.

Chapter 12: Creating custom metadata

The Data Catalog component is a utility component used by ION Desk.

This section explains the Data Catalog contents and how you can add metadata for your own objects or for extensions on standard application objects.

The Data Catalog contains metadata on objects that are sent through the ION Service. During the installation of ION, the Data Catalog is filled with the latest version of the Infor metadata.

If you only use standard objects from Infor, you do not have to modify the Data Catalog contents. You can use custom application objects in ION Connect, in a monitor, or in a workflow activation policy. Add the metadata for those custom application objects to the Data Catalog.

The functionality to add custom metadata in the Data Catalog is intended only for customer-specific metadata. Infor metadata should not be included as custom metadata.

Data Catalog contents

The Data Catalog contains noun metadata information organized in libraries. A library can contain one or more nouns. Libraries contain standard objects, which can either be global or application-specific. You can use the available noun metadata from all libraries in ION Desk models.

Custom objects can be of type BOD, also referred to as "custom nouns", or of type ANY, DSV, or JSON.

At each library level, each tenant level, and each noun level, a patch number is maintained. Only the latest noun metadata for both standard and custom nouns is stored in the Data Catalog.

For each noun, custom or standard, this information is stored:

- Noun Name
- Noun XSD - the XML schema definition for this noun, flattened
- Other metadata, such as the patch number (a kind of version number), the list of XPaths to key fields in the noun, the list of verbs supported by this noun, and the relationships between this noun and other nouns.

For custom objects of type ANY, only the object name is stored.

For custom objects of type JSON, the JSON schema definition for each object name is stored.

For custom objects of type DSV, the DSV schema definition for each object name is stored.

Before customizing the Data Catalog

Before you customize the Data Catalog, note these points:

- Contact Infor for information on services to assist you when developing custom objects.
- The Data Catalog contains these files:
  - XML Schema (XSD) files and XML files to describe objects of type BOD
  - JSON Schema files to describe objects of type JSON and DSV (delimiter-separated values)

  To customize the Data Catalog metadata, you must understand and adapt those files.
- If you add files to the Data Catalog, you can use customized nouns in ION Desk only. To use the customized nouns in ION Service, these nouns must be adopted by Infor applications or other applications. The definitions in the Data Catalog must always match the objects that are sent or received by applications that are connected to ION.
- In the Data Catalog XSD files, conventions are used. These are the most important conventions:
  - All elements use the type-attribute and therefore refer to named types.
  - All types, complex and simple, are explicit and have "Type" as a post fix. For example, `Status Type`.
  - The elements in a complexType containing a sequence or choice are always a `ref` to a single element with a named type.
- **Note:**
  - In customized XSDs, do not use unions such as `<xs:union memberTypes="xs:string xs:dateTime"/>` or binary data types such as `xs:hexBinary`.
  - If you create a customized version of a standard noun, it must be compatible with the standard. If you use a new name for the customized noun, such as MySalesOrder, compatibility with the standards is not required. When you create a version of your custom noun, it must be backwards compatible with previous versions.

Object Naming Conventions

Objects of all types are registered by name.

These constraints apply:

- The object name must be unique across all custom objects types and standard nouns and application specific nouns. For example, if the `InforOagis` library is imported, a custom object with the name `Sync.SalesOrder` must not exist.
- The object name is case-preserving, but case-insensitive. If 'MyDocument' exists, you cannot create 'MyDOCUMENT' except by overwriting the existing one.
- The maximum length for an object name is 100 characters.
- The object names for the metadata files cannot exceed 255 characters. This is including the path in the export file, for example: `JSON/myObject/myObject.schema.json`

Custom objects of type ANY

You can use objects of type ANY in ION File Connector and IMS Connector.

A custom object of type ANY is only defined by its name.

An object name may contain these characters:

- Any standard letter in any language
- Numbers 0-9
- Underscore (_)
- Hyphen (-)
- Period (.)

Defining a custom object of type ANY

1 To create an object of type ANY, prepare this folder structure for the import:

FOLDER: `ANY`

+-- FOLDER: *`<object name>`*

+---- FILE: *`<object name>`*`.xml`

The *`<object name>`*`.xml` file is empty for objects of type ANY.

2 To import a ZIP file containing one or more definitions for objects of type ANY, use the **Custom Objects** page in ION Desk.

See the *Infor ION Desk User Guide*.

Custom objects of type JSON

You can use objects of type JSON in ION File Connector, IMS Connector, and ION AnySQL Connector.

A custom object of type JSON is defined by its name and a JSON schema file.

An object name may contain these characters:

- Any standard letter in any language
- Numbers 0-9
- Underscore (_)
- Hyphen (-)
- Period (.)

Defining a custom object of type JSON

1 To create an object of type JSON, prepare this folder structure for the import:

FOLDER: `JSON`

+-- FOLDER: *`<object name>`*

+---- FILE: *`<object name>`*`.schema.json`

+---- FILE: *`<object name>`*`.properties.json`

2 To import a ZIP file containing one or more definitions for objects of type JSON, use the **Custom Objects** page in ION Desk.

See the *Infor ION Desk User Guide*.

These validations are performed upon import:

- The object name must be unique and must comply to the object naming restrictions mentioned above.
- If an object with the same name is already registered with another type, the import fails.
- If an object with the same name is already registered with type JSON, the imported schema overwrites the existing schema.
- The *`<object name>`*`.schema.json` schema file is validated as follows:
  - There must be only one JSON schema file for a given object name.
  - The schema file must be valid JSON according to JSON schema definition, version draft-06.
  - The schema file must use UTF-8 encoding.
  - The “anyOf” keyword is not allowed.
  - The “oneOf” keyword is supported only to describe a type that can be null.
- The *`<object name>`*`.properties.json` file is optional and can contain additional metadata properties for the object.

  For information on how to define this file, see Defining additional object metadata properties on page 105 and Additional properties file on page 105.

Newline-delimited JSON

If your data object uses newline-delimited JSON, you must specify that in your object schema.

1 Specify the schema for a single object.

All objects in a single data object must have the same schema.

2 Include a property called `x-stream` and set it to true.

This code is a newline-delimited JSON schema example:

```
{
  "$schema": "http://json-schema.org/draft-06/schema#",
      "x-stream":true,
      "type": "object",
      "properties": {
        "field1": {
```

```
          "type": "integer"
        },
        "field2": {
          "type": "string"
        }
    }
```

This code shows the JSON data that corresponds with the schema example:

```
{"field1":123,"field2":"Some text"}
{"field1":456,"field2":"Another text"}
{"field1":789,"field2":"More text to be added"}
```

Validations that are performed on import for newline-delimited JSON follow the same rules as conventional JSON.

Custom objects of type DSV

You can use objects of type DSV in ION File Connector and IMS Connector. DSV stands for Delimiter-Separated Values.

When you register a DSV object schema in the Data Catalog, it is assigned a subtype. Based on the separator value that is provided in the schema, one of these subtypes is assigned:

- CSV - comma-separated values
- TSV - tab-separated values
- PSV - pipe-separated values
- Other

A custom object of type DSV is defined by its name and a schema file.

An object name may contain these characters:

- Any standard letter in any language
- Numbers 0-9
- Underscore (_)
- Hyphen (-)
- Period (.)

Defining a custom object of type DSV

1 To create an object of type DSV, prepare this folder structure for the import:

FOLDER: `DSV`

+--FOLDER: *<object name>*

+----FILE: *<object name>*`.schema.json`

+----FILE: *<object name>*.properties.json

2 These validations are performed upon import:

- The object name must be unique and must comply to the object naming restrictions mentioned above.
- If an object with the same name is already registered with another type, the import fails.
- If an object with the same name is already registered with type DSV, the imported schema overwrites the existing schema.
- The *<object name>*.schema.json schema file is validated as follows:
  - There must be only one DSV schema for a given object name.
  - The schema file must use UTF-8 encoding.
  - The schema file must contain a “dialect” property. This property defines the format of the delimited data object.

    The dialect property must contain a “separator” property, which defines the character that separates the values in a row within the data object.
  - The “properties” element in the schema file should list the fields, in order, that are expected to be found in the DSV data object. This section of the schema must be valid JSON according to schema definition, version draft-06.
    - The “anyOf” keyword is not allowed.
    - The “oneOf” keyword is supported only to describe a type that can be null.
- The *<object name>*.properties.json file is optional and can contain additional metadata properties for the object.

  For information on how to define this file, see Defining additional object metadata properties on page 105 and Additional properties file on page 105.

This code is an example of a DSV schema definition:

```
{
   "title": "myDelimitedFile",
   "description": "DSV Schema for myDelimitedFile",
   "dialect": {
      "separator": ",",
      "skipLines": 1,
      "headerLine": 1,
      "enclosingCharacter": "\""
   },
   "properties": {
      "Code": {
         "description": "Customer code",
         "type": "string",
         "maxLength": 10
      },
      "Customer": {
         "description": "Customer name",
         "type": "string",
         "maxLength": 100
      },
      "PubDatetime": {
         "description": "Publication timestamp",
         "type": "string",
```

```
          "format": "date"
        }
      }
    }
```

Dialect properties for DSV objects

. This table shows the dialect properties that are supported for DSV objects.

Dialect property	Type	Description
`separator`	string	The delimiter character that separates the values in a row of data. The value of the separator should be defined as follows for the different delimited file types: • Comma-separated (CSV): `"separator":","` • Tab-separated (TSV): `"separator":"\t"` • Pipe-separated (PSV): `"separator":"|"` Other separator values are allowed, but these must be single-character separators. For example, double pipes (`||`) are not supported. Required
`skipLines`	integer	Indicates the number of header rows to skip over at the top of the file before reaching the actual data. Optional **Note:** For Compass (Data Lake Services) only, DSV data objects are required to have a single header line containing the column names. `skipLines` is always assumed to be 1.
`headerLine`	integer	Indicates the line number that contains the column headers for the data object. This value must be less than or equal to the value of `skipLines`. Optional **Note:** For Compass (Data Lake Services) only, DSV data objects are required to have a single header line containing the column names. `headerLine` is always assumed to be 1.
`enclosingCharacter`	string	The character that identifies the start and end of a value. If two consecutive enclosing characters are found in a data object, they are interpreted as one, therefore escaping the enclosing character. Optional

The line separator is not specified in the metadata.

These characters are regarded as the end of a line, unless they are placed within enclosing characters:

- A carriage return
- A line feed
- The combination of carriage return and line feed

The last line may or may not have a line separator.

The encoding is not specified in the metadata. It is always assumed to be UTF-8.

Metadata for localized strings

A localized string is a value that can be represented in different languages, with each value identified in the context of a locale.

For example, if your data object contains a field called “description,” this field can have a string value representing the English translation and another string value representing the Spanish translation. To ensure that localized strings are properly handled within your data objects, you must define them as localized in your object schema.

There are two methods in which you can define a property as localized in the object metadata.

Method 1

This method can only be used for JSON objects

To specify that a field or property contains localized data:

1. The localized property type must be set to “object”.
2. Include a Boolean property called `x-localized` and set it to true.
3. To define a maxLength for each of the localized strings, include a property called `x-properties MaxLength`.

   For information on localized strings in data objects, see Data object definitions for localized string values on page 134.
4. Specify the maximum string length.
5. In your localized property definition, you must set the `additionalProperties` property to true.

   Note: This property is not required within your definition since the default is true when this property is not specified.

JSON schema example (Method 1)

This is a JSON Schema example for data objects containing localized strings, using method 1:

```
{
   "$schema":"http://json-schema.org/draft-06/schema#",
```

```
   "type":"object",
   "properties":{
      "id":{
         "description":"The identifier",
         "type":"string",
         "maxLength":40
      },
      "description":{
         "description":"Additional information",
         "type":"object",
         "x-localized":true,
         "x-propertiesMaxLength":250,
    "additionalProperties":true
      }
   }
}
```

Inside the data objects, localized properties must include the supported locales, along with their associated string values.

This example of a data object relates to the JSON Schema example for data objects containing localized strings.

```
{"id":"123","description":{"en_US":"car","es_ES":"coche"}}
```

The description property contains values for two locales; one for `en_US` and another for `es_ES`.

Method 2

This method can only be used for both JSON and DSV objects.

To specify that a field or property contains localized data:

1. Include a Boolean property called `x-localized` and set it to true.
2. Specify a maximum length for the localized strings using maxLength.

   For information on localized strings in data objects, see Data object definitions for localized string values on page 134.

DSV schema example (Method 2)

This is a DSV Schema example for data objects containing localized strings, using method 2:

```
{
   "title":"myDelimitedFile",
   "description":"Sample schema for DSV with localized data",
   "dialect":{
      "separator":",",
      "skipLines":1,
```

```
      "enclosingCharacter":"\""
   },
   "properties":{
      "id":{
         "description":"The identifier",
         "type":"string",
         "maxLength":40
      },
      "description":{
         "description":"Additional Information",
         "type":"string",
         "x-localized":true,
         "maxLength":250
      }
   }
}
```

Inside the data objects, localized properties must contain a separate column for each of the supported locale codes. The locale code must be used as a suffix to the property name.

This example of a data object relates to the DSV Schema example for data objects containing localized strings.

```
"id","description_en_US","description_es_ES"
"123","car","coche"
```

Using datetime formats

When dates and times are included in JSON and DSV data objects, you should define them as such in the object metadata. This makes it clear to applications that use the metadata that the values should be interpreted as dates or times. Infor applications that use dates and times must follow the ISO 8601 RFC 3339 standard formats.

JSON Schema draft-06 supports these date and time formats for string instances:

- date-time
- date
- time

When you include dates and times in your objects, it is best practice to use “date-time”, in UTC, wherever possible. Dates without a time and times without a date regularly occur in application data. Therefore, you can also use the "date" or "time" formats alone.

Standard format definitions

This table shows the standard format definitions:

Metadata definition	Expected format in data object	Example data
`"datetimeProperty":{ "type":"string", "format":"date-time" }`	yyyy-MM-ddTHH:mm:ss[.S]Z	"2017-07-04T14:08:43Z" Or "2017-07-04T14:08:43.123Z"
`"dateProperty":{ "type":"string", "format":"date" }`	yyyy-MM-dd	“2017-07-04”
`"timeProperty":{ "type":"string", "format":"time" }` This format is not supported by Data Lake services when processing data. Times without dates are treated as string values.	HH:mm:ss[.S]	“14:08:43” Or “14:08:43.123”

Custom datetime formats

To define a datetime format that is not a standard supported by JSON schema draft-06, you can use the “x-dateTimeFormat” custom property in your object metadata. This enables the interpretation of data from third-party applications that do not adhere to the ISO 8601 RFC 3339 standard.

Custom format definitions

The tables in this section show the custom datetime formats that are currently supported. Any other formats specified are not recognized by applications that use the metadata. All dates that are included in a data object are assumed to be in UTC. The value that is provided for “x-dateTimeFormat” is case-sensitive. Therefore, you must include this value in your metadata exactly as it is defined below.

This table shows the metadata definitions:

Metadata definition	Description

<pre>"epochMillisProperty":{
"type":"integer",
"x-dateTimeFormat":"epoch-millis"
}</pre>
	Datetime in epoch milliseconds for integer instances.
	Example: 1537204639000

<pre>"americanDateTimeProperty":{
"type":"string",
"x-dateTimeFormat":"M/d/yyyy h:mm:ss[.S] a"
}</pre>
	American datetime format for string instances.
	Examples:
	"10/9/2017 12:42:01 PM"
	or
	"10/9/2017 12:42:01.139 PM"

<pre>"americanDateProperty":{
"type":"string",
"x-dateTimeFormat":"M/d/yyyy"
}</pre>
	American date format for string instances.
	Example: "6/23/2018"

<pre>"integerDateProperty":{
"type":"integer",
"x-dateTimeFormat":"yyyyMMdd"
}</pre>
	8-digit integer date for integer instances.
	Example: 20180601

<pre>"threeCharMonthProperty":{
"type":"string",
"x-dateTimeFormat":"dd-MMM-yyyy HH:mm:ss"
}</pre>
	Three-character month for string instances, Oracle format. This format always translates to a US locale.
	Example: "01-SEP-2018 14:08:23"

This table shows the tokens for datetime formats:

Token	Description
M	Month, one or two digits: 1-12
MM	Month, two digits: 01-12
MMM	3-character month: JAN, FEB, MAR, etc.
D	Day of the month, one or two digits: 1-31
dd	Day of the month, two digits: 01-31
yyyy	Year, four digits: 0001-9999
HH	Hours, 24-hour, two digits: 00-23
h	Hours, 12-hour, one or two digits: 1-12
mm	Minutes, two digits: 00-59
ss	Seconds, two digits: 00-59

Token	Description
[S]	Optional fractional seconds, up to nine digits: 0-999999999
a	AM/PM designator when using 12-hour format: "AM", "am", "PM", or "pm"

Schema Property Order

Standard JSON Schema does not support specifying an order to the properties defined in a schema. If the fields in your JSON or DSV data objects must be displayed in a specific order, you can use the custom property “x-position” to define what that order must be. The value for "x-position" must be an integer.

Example Schema:

```
{
   "$schema":"http://json-schema.org/draft-06/schema#",
   "title":"Schema Property Order",
   "description":"Sample schema to include property order",
   "type":"object",
   "properties":{
       "divisionName":{
            "type":"string",
            "maxLength":16,
        "x-position":2
         },
         "divisionId":{
            "type":"string",
            "maxLength":250,
        "x-position":1
         },
         "updateDateTime":{
            "type":"string",
            "format":"date-time"
         },
         "variationNumber":{
            "type":"integer"
         },
         "companyId":{
            "type":"string",
            "x-position":3
         }

   }
```

Since “x-position” is optional, it is not required to include it for all properties in the schema. It is up to the applications using this metadata to determine how to handle the display of fields that do not include an “x-position” value.

Defining additional object metadata properties

When you import a schema, you can register additional metadata properties for objects of type JSON and DSV with the Data Catalog.

To register these additional metadata properties, you require an object properties file.

This is an optional file that contains extra properties that are not defined in the object schema. The file name must match the object name and have the `.properties.json` extension.

You can include these additional properties when you import the object schema in the Data Catalog UI. For information on the file structure for including additional metadata properties, see the "Zip file structure for import" section in the *Infor ION Desk User Guide*.

Additional properties file

Validation

The properties file is validated using this schema:

```
{
  "$schema": "http://json-schema.org/draft-06/schema#",
  "title": "Additional Object Metadata",
  "description": "Additional Object Metadata",
  "type": "object",
  "properties": {
    "IdentifierPaths": {
      "type": "array",
      "items": {
        "type": "string"
      }
    },
    "VariationPath": {
      "type": "string"
    },
    "TimestampPath": {
      "type": "string"
    },
    "DeleteIndicator": {
      "type": "object",
      "properties": {
        "path": {
          "type": "string"
        },
        "value": {
          "type":["string","boolean","number"]
        }
      },
      "additionalProperties":false,
      "required":["path","value"]
    },
```

```
    "AdditionalProperties": {
      "type": "object",
      "additionalProperties": true
    }
  },
  "additionalProperties": false
}
```

Properties

This table shows each of the properties that may be included in the file. All properties are optional.

Property	Description
`IdentifierPaths`	The set of properties of the data object that identify the object. This applies if the data object consists of a single JSON or DSV object. Alternatively, the set of properties of the data objects. This applies if the data object holds a newline-delimited list or an array of objects. Each property in the array must be a JSON path to the property that is the identifier or part of the identifier.
`VariationPath`	The path to a property that indicates a variation number or variation string. This must be a JSON path. If this property is available, it can be used for these purposes: • To determine the sequence in which multiple changes on a single object took place • To find the latest version of an object
`TimestampPath`	The path to a property that indicates the moment the object was last updated or created in the application that owns the data. This must be a JSON path. The property it refers to must have a date-time format. If this property is available, it can be used to determine the order in which changes on a set of objects from the same application took place. This property is less accurate than `VariationPath`, but can be used across objects.
`AdditionalProperties`	This area can be used for additional properties that are not owned or prescribed by the Data Catalog. Any custom properties for applications must be added here. **Note:** When you add additional properties, ensure that the name of each property is unique. We recommend that you include the application logical ID as a prefix to the property name.

Property	Description
`DeleteIndicator`	The path to the property that indicates whether the object is deleted, or marked as deleted, and the property value to indicate that. The property must be a boolean, number, or string, and the specified value must match that data type. When you use `DeleteIndicator`, these properties are required: • path - The path to a property that holds the delete indicator for the object. This must be a JSON path. • value - The value of the delete indicator property to determine that the object is deleted. This value can be a string, boolean, or number. The value must match the data type of the property that “path” is pointing to.

Example

This code shows an example of an object’s properties file:

```
{
   "IdentifierPaths":[
      "$.*.myIdProperty",
      "$.*.mySecondIdProperty"
   ],
   "VariationPath":"$.*.myVariationId",
   "TimestampPath":"$.*.myTimestampProperty",
   "DeleteIndicator":{
      "path":"$.*.myDeleteIndicatorProperty",
      "value":"deleted"
   },
   "AdditionalProperties":{
      "infor_ies_searchPath":{...},
      "infor_ies_indexDefinition":{...},
      "acme_myCustomThingies":{...}
   }
}
```

Defining a custom noun

You can add the metadata for your own nouns to the Data Catalog.

Customer-defined nouns do not have to fully, adopt all practices as used in standard nouns, such as the way object IDs or references are formatted. They must meet these conditions:

- The noun content is in XML.
- The noun has an identifying attribute.
- The noun's envelope is a BOD.

Nouns are described by XML Schema (XSD) files. The Infor-delivered nouns are the officially published schemas. Customers can create their own XMLSchema for their nouns.

The name of a custom noun must be unique compared to the standard noun names that are already uploaded in the Data Catalog. We recommend that you always precede Custom nouns with "My. " For example, `MyMaterialRelease` or `MyShippingSchedule`. Using "My "avoids naming conflicts with standard Infor nouns.

The name of a custom noun may contain:

- Any standard letter in any language
- Numbers 0-9
- Underscore

To add a custom noun to the Data Catalog, you must prepare an archive file containing the metadata. You can upload several custom nouns in one archive at the same time. For each custom noun, a noun data-set composed of two files must be present and must contain the noun name in the title:

- `<nounname>.xml`
- `<nounname>.xsd`

These additional validations are performed on the archive file upon import by ION Desk:

- The archive may only contain valid XSD and valid XML files.
- The noun name of the XSD file must be the same as the noun name used inside the XSD file.
- The custom noun definition may only be defined in one XSD file.
- The XML files that defined the metadata of the custom noun must refer to this namespace:
  `xmlns="http://schema.infor.com/CloudRegistry/1"`

Upon upload of a custom noun, ION Desk assigns this noun a patch number, depending on the first available number for the current tenant. When the same custom noun is uploaded again, the patch number increases. ION Desk does not compare the noun definition contents of the new noun and the noun that already exists in the Data Catalog.

To create the metadata for a custom noun:

1 Define the custom noun schema file (XSD) and verify that this is a valid schema. Save this file with the name `<nounname>.xsd`.

Complete these steps:

a Copy the XMLSchema text below into a file with the name of your noun with a "`.xsd`" extension. For example: `MyMaterial.xsd`.

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema targetNamespace="http://schema.infor.com/InforOAGIS/2"
xmlns="http://schema.infor.com/InforOAGIS/2"
xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qual
ified" attributeFormDefault="unqualified" version="1.0.0">
   <xs:element name="MySampleDocument" type="MySampleDocumentType"/>

   <xs:complexType name="MySampleDocumentType">
      <xs:sequence>
         <xs:element name="SampleHeader" type="SampleHeaderType"
minOccurs="1"/>
         <xs:element name="SampleLine" type="SampleLineType"
minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
```

```
   </xs:complexType>
   <xs:complexType name="SampleHeaderType">
      <xs:sequence>
         <xs:element name="DocumentID" type="xs:normalizedString"
minOccurs="0"/>
         <xs:element name="DocumentDateTime" type="xs:dateTime"
minOccurs="0"/>
        <xs:element name="Status" type="xs:string" minOccurs="0"/>

         <xs:element name="Description" type="xs:string" minOc
curs="0"/>
         <xs:element name="ShipmentID" type="xs:normalizedString"
minOccurs="0"/>
         <xs:element name="IsActive" type="xs:boolean" minOc
curs="0"/>
       </xs:sequence>
   </xs:complexType>
   <xs:complexType name="SampleLineType">
      <xs:sequence>
         <xs:element name="LineNumber" type="xs:integer"/>
        <xs:element name="Amount" type="xs:decimal" minOccurs="0"/>

         <xs:element name="Note" type="xs:string" minOccurs="0"
maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
</xs:schema>
```

b Open the file in an editor and replace all occurrences of "`MySampleDocument` " with "`MyMaterial`".

c Adapt the file as required to define the attributes of the custom noun. Do not refer to XSD files from the Standard folder.

2 Define the custom noun metadata properties file with the name `<nounname>.xml`. Create a file named `[NounName].xml`. For example, if your custom noun is called `MyMaterial`, the file must be named `MyMaterial.xml`.

In this file, add an entry to specify the identifier for the custom noun, verbs that are supported, and the relation it has with other nouns. Note the ‘IDXPath’ is mandatory, you can leave the ‘Relation’ element empty if there is no other noun to reference.

For example:

```
<?xml version="1.0" encoding="utf-8"?>
<NounMetadata xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://schema.infor.com/InforOAGIS/2">
   <Noun>
      <NounName>MySampleDocument</NounName>
      <NounType>Transactional</NounType>
     <IDXPath>/*/DataArea/MySampleDocument/SampleHeader/DocumentID</IDX
Path>
      <DescriptionXPath>/*/DataArea/MySampleDocument/SampleHeader/De
scription</DescriptionXPath>
     <StatusXPath>/*/DataArea/MySampleDocument/SampleHeader/Status</Sta
```

```
tusXPath>
<DocumentDateTimeXPath>/*/DataArea/MySampleDocument/SampleHeader/Docu
mentDateTime</DocumentDateTimeXPath>
      <SupportedVerbs>
          <SupportedVerb>Acknowledge</SupportedVerb>
          <SupportedVerb>Process</SupportedVerb>
          <SupportedVerb>Sync</SupportedVerb>
      </SupportedVerbs>
   </Noun>
   <Relation type="Transactional">
      <ToNoun>Shipment</ToNoun>
      <Priority>10</Priority>
     <RelationLabel>My sample document linked to shipment</RelationLa
bel>
      <RelationPaths>
         <FromNounPath>/*/DataArea/MySampleDocument/SampleHeader/Ship
mentID</FromNounPath>
          <ToNounPath>/*/DataArea/Shipment/ShipmentHeader/Documen
tID[1]/ID</ToNounPath>
      </RelationPaths>
   </Relation>
</NounMetadata>
```

3 Create a folder for each custom noun and place corresponding xsd schema file and xml file underneath it. The folder name must match the noun name.

4 Gather the custom noun metadata files into one archive file and import this archive into ION Desk. For information about importing custom nouns into ION Desk, see the *Infor ION Desk User Guide*.

Once the custom noun is uploaded to the Data Catalog, it is visible to all users of the current tenant. You can perform these actions using this custom noun:

- Select the application object in a connection point.
- Select the attributes of the object in a filter or content-based routing in an object flow.
- Select the application object and its attributes in a workflow activation policy.
- Select the application object, its references, and its attributes in a monitor.

Important notes:

When using namespaces in your XSDs, the local element name must be unique at any level in a BOD. Two elements having the same name but a different namespace must not exist within the same parent element. So, for example, this object is not supported:

```
<Customer>
  <Address xmlns="namespace1">
    …
  </Address>
  <Address xmlns="namespace2">
    …
  </Address>
</Customer>
```

Customizing an existing noun

Standard nouns contain placeholders for adding customizations: the UserAreas. Usually, UserArea elements are available in multiple locations for a noun. For example, in the header and in the lines.

You can use the UserArea in two ways:

- Using properties in the UserArea. This is the preferred method, because it is supported automatically for any BOD. No changes are required in the Data Catalog.
- Using a custom XML structure in the UserArea. This method is required if the custom data is too complex to fit in the property structure.

Both approaches are explained below.

Using properties in the UserArea

If your custom data can be organized in name-value pairs, you can use the standard 'Property' structure to include this data in the UserArea. In that case customize the application that sends a BOD so that it includes the required properties. Customize the application that receives the BOD so it can handle the data.

This code is an example of a UserArea containing data using the standard Property structure:

```
<UserArea>
   <Property>
      <NameValue name="AcmeCustomNote" type="StringType">My
note</NameValue>
   </Property>
   <Property>
      <NameValue name="AcmeCustomQuantity" type="Numeric
Type">10.00</NameValue>
   </Property>
</UserArea>
```

Note: To avoid name clashes with properties that may already be used by some applications, we recommend that you use a prefix, such as your company name.

This table shows some of the types that you can use:

Type	Examples of Values	
StringType	This is a string	
NumericType	10.00	
IntegerNumericType	1234	
DateTimeType	2004-12-31T12:32:14.123Z	
IndicatorType	true, false	Two possible values

To use UserArea properties in ION, no changes are required in the Data Catalog. If you use a BOD that has a UserArea, you can select your properties in an event monitor or an activation policy.

Note: You cannot select your properties in ION Connect (content-based routing and filtering).

To select a UserArea property:

1. In the attribute selection window, find the location of the User Area and expand the User Area.

   For example, the window can contain this code:

```
Contract
 ContractHeader
  …
  UserArea
   Property
    NameValue
```

2. Select the NameValue attribute.
3. Specify the data type for the attribute.

   Normally the data type is retrieved from the Data Catalog but when using a UserArea Property you must specify the data type, because properties can have different data types.
4. For the selected NameValue attribute, define an attribute filter on the `name` to select the specific property you are interested in.

   For example:

   `Contract/ContractHeader/UserArea/Property/@name = "AcmeCustomQuantity"`

   Note: If you use multiple properties from the same user area in an event monitor or workflow activation policy, you must define the filter for the first property before you can select the next property. Otherwise two selected attributes will have the same XPath, which is not accepted in ION Desk.

Using a custom XML structure in the UserArea

If your custom data is too complex to fit in the `standardProperty` structure, you must use a custom XML structure.

By default, the `UserArea` elements in BODs are of type 'AnyType', because they can contain any data: standard properties, your custom structure, or anything else. In these situations, you must define an XSD in the Data Catalog:

- To select your properties in ION Connect, in content-based routing or filtering.
- To select your properties in an event monitor or an activation policy.

After adding your XSD to the Data Catalog as described later, your custom elements are displayed in the attribute selection windows in ION Desk.

For example, assume you must add this custom data in the BOD UserArea:

```
<UserArea>
    <AcmeCustomContacts>
        <Contact>
            <Name>Someone</Name>
```

```
            <Email>someone@acme.com</Email>
        </Contact >
        <Contact>
            <Name>Someone Else</Name>
            <Email>someone.else@somewhere.com</Email>
        </Contact >
    </AcmeCustomContacts >
</UserArea>
```

In this case, you must complete these steps:

1. Customize the connection point that sends the BOD, to fill the `UserArea` as required. Customize the connection point that receives the BOD, to handle the data.
2. Create an XML Schema Definition (XSD) file to describe the added XML.

   For example, the `AcmeCustomContacts.xsd` file can contain this code:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
 elementFormDefault="qualified" attributeFormDefault="unqualified">
    <xs:element name="AcmeCustomContacts" type="AcmeCustomContact
sType"/>
    <xs:complexType name="AcmeCustomContactsType">
        <xs:sequence>
          <xs:element name="Contact" type="ContactType" maxOccurs="un
bounded"/>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="ContactType">
        <xs:sequence>
            <xs:element name="Name" type="xs:string"/>
            <xs:element name="Email" type="xs:string"/>
        </xs:sequence>
    </xs:complexType>
</xs:schema>
```

   Note: The XSD for a user area extension must have a single top-level element. To add multiple elements, put them under a single parent element.

3. In ION Desk, select **Configure > User Area Extensions > Schema Files** and add the XSD file to the Data Catalog.
4. In ION Desk, select **Configure > User Area Extensions > Extensions**. Map the schema file to the UserArea elements of the nouns that use the User Area Extension.

Note: For details about the **Configure > User Area Extensions** pages, see the *Infor ION Desk User Guide*.

Using an XSD extension for validation

By default, the UserArea is an 'AnyType', because the UserArea can contain any data: standard properties, your custom structure, or anything else. In this default situation, a BOD normally validates

successfully against the BOD XSD, independent of the contents of the UserArea. If you defined a UserArea extension XSD in the Data Catalog, to use the XSD for validating your BOD XML, you must ensure the validator can find the XSD extension.

To achieve this:

1 Add namespace information to the XSD.
For example, the `AcmeCustomContacts.xsd` file can contain this code:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns="http://schema.acme.com/userarea"
  targetNamespace=="http://schema.acme.com/userarea"
  elementFormDefault="qualified" attributeFormDefault="unqualified">
    <xs:element name="AcmeCustomContacts" type="AcmeCustomContact
sType"/>
    <xs:complexType name="AcmeCustomContactsType">
        …
    </xs:complexType>
</xs:schema>
```

2 In the XML that contains the custom data, include a reference to the namespace:

```
<UserArea>
    <acme:AcmeCustomContacts xmlns:acme="http://schema.acme.com/user
area">
        <acme:Contact>
             <acme:Name>Someone</acme:Name>
             <acme:Email>someone@acme.com</acme:Email>
         </acme:Contact >
     </acme:AcmeCustomContacts >
</UserArea>
```

Note: The namespace, in this case "`http://schema.acme.com/userarea`", must be the same as in the XSD.

After completing these steps you can validate the BOD XML against the BOD XSD. In this case, for example, validate the BOD XML against `SyncContract.xsd`.
Note: Validation can help when you set up and test a customization, but reduces performance. Therefore, if possible, avoid validation of all messages sent or received by your application. The ION Service also does not validate the BODs against the XSDs as defined in the Data Catalog.

Chapter 13: Custom message headers

You can define custom message headers for objects that are stored in the Data Catalog. After you have defined custom message headers, they are available to attach to objects that are processed by ION services.

To define custom headers, you can use one of these methods:

1. Direct import to Data Catalog through ION Desk. On the **Object Schemas** page, you can include a custom headers file in the object import ZIP file. This method is useful if you only want to add or update custom headers for an object, but not the object itself. It is also useful when updating custom headers for multiple objects in the Data Catalog.
2. Include custom headers when adding or updating an object using the Data Catalog POST /v1/object API.
   Note: This option is not viable for BODs because the API only supports JSON, DSV, and ANY objects.
3. In the Data Catalog application on an object's details page.

For more information on methods 1 and 3, see the "Configuring custom message headers" section in the *Infor ION Desk User Guide*.

Custom headers file format

ZIP file import validation

When you import custom headers in ION Desk, the file is validated using this schema:

```
{
  "$schema": "http://json-schema.org/draft-06/schema#",
  "$id": "http://json-schema.org/draft-06/schema#",
  "title": "Custom Headers Schema",
  "type": "object",
  "properties": {
    "customHeaders": {
      "type": "array",
      "items": {
        "type": "object",
        "properties": {
          "objectName": {
            "type": "string"
```

```
          },
          "headers": {
            "type": "array",
            "maxItems": 3,
            "items": {
              "type": "object",
              "properties": {
                "name": {
                  "type": "string",
                  "pattern": "^Custom_[a-zA-Z0-9]+$",
                  "maxLength": 250
                },
                "dataType": {
                  "enum": [
                    "String",
                    "Integer",
                    "Decimal",
                    "Boolean",
                    "Date",
                    "DateTime"
                  ]
                }
              },
              "required": [
                "name",
                "dataType"
              ]
            }
          }
        },
        "required": [
          "objectName",
          "headers"
        ]
      }
    }
  }
}
```

This is an example of a custom headers file:

```
{
  "customHeaders": [
    {
      "objectName": "Process.PlannedTransfer",
      "headers": [
        {
          "name": "Custom_processItemID",
          "dataType": "String"
        }
      ]
    },
    {
      "objectName": "Sync.PlannedTransfer",
```

```
      "headers": [
        {
          "name": "Custom_syncItemID",
          "dataType": "String"
        },
        {
          "name": "Custom_syncUPCID",
          "dataType": "String"
        }
      ]
    },
    {
      "objectName": "Inventory_Repo",
      "headers": [
        {
          "name": "Custom_itemNumber",
          "dataType": "Integer"
        }
      ]
    },
    {
      "objectName": "MITMAS",
      "headers": [
        {
          "name": "Custom_attributeModel",
          "dataType": "String"
        }
      ]
    }
  ]
}
```

Data Catalog API validation

When you register an object with the Data Catalog through the POST /v1/object API, custom headers for that object are validated against this schema:

```
{
  "$schema": "http://json-schema.org/draft-06/schema#",
  "$id": "http://json-schema.org/draft-06/schema#",
  "title": "Custom Headers Schema",
  "type": "object",
  "properties": {
    "customHeader": {
      "type": "object",
      "properties": {
        "objectName": {
          "type": "string"
        },
        "headers": {
          "objectName": {
            "type": "string"
          },
```

```
          "headers": {
            "type": "array",
            "maxItems": 3,
            "items": {
              "type": "object",
              "properties": {
                "name": {
                  "type": "string",
                  "pattern": "^Custom_[a-zA-Z0-9]+$",
                  "maxLength": 250
                },
                "dataType": {
                  "enum": [
                    "String",
                    "Integer",
                    "Decimal",
                    "Boolean",
                    "Date",
                    "DateTime"
                  ]
                }
              },
              "required": [
                "name",
                "dataType"
              ]
            }
          },
          "required": [
            "objectName",
            "headers"
          ]
        }
      }
    }
  }
}
```

This is an example of the input body for the API that is used to register an object with custom headers:

```
{
  "name":"mySalesOrder",
  "type":"JSON",
  "schema": {...},
  "properties":{...},
  "customHeader": {
        "objectName": "mySalesOrder",
        "headers": [
            {
                "name": "Custom_salesOrderId",
                "dataType": "Integer"
            },
            {
                "name": "Custom_shipToAddress",
```

```
                "dataType": "String"
            }
        ]
    }
}
```

Properties

This table shows the properties in the file:

Property	Description
customHeaders	An array of JSON objects. Each JSON object contains the name of an object schema and the headers to be defined for that specified object. Optional. **Note:** Applies to the zip file import only.
CustomHeader	A JSON object that contains the object name and the list of headers to be defined for that object. Optional. **Note:** Applies to the POST API only.
objectName	The name of the object for which the custom headers should be defined. For BODs, you must specify the full object name as opposed to only the noun. For example, `Sync.SalesOrder`. Required.

Property		Description
headers		An array of headers to be defined for the object. A maximum of three custom headers may exist per object. Required, but the array can be empty. If an empty array is found, any existing custom headers for the associated object are deleted.
	name	The name of the custom header. The name must begin with `Custom_` and can only contain alphanumeric characters, without any spaces. It cannot exceed 250 characters in length. Required.
	dataType	The data type that the custom header value is expected to have. These are possible data types: • String • Integer • Decimal • Boolean • Date • DateTime **Note:** These are case-sensitive and must match exactly to pass validation. Required.

Chapter 14: Application Programming Interface (API)

You can use several ION APIs.

The available ION APIs are:

- ION OneView
- Alarms
- IMS
- ION Process
- Data Catalog
- Business Rules
- Data Lake

For more information on how to use the APIs, see the *Infor ION API Administration Guide*.

ION services such as OneView can only be used through ION API. As an ION user you must be authorized as administrator to see the ION API icon in the Infor Ming.le application menu. Click the ION API icon to find details such as the existing methods.

ION OneView API

ION OneView is used for troubleshooting of integrations and for tracing of data objects throughput in ION.

ION OneView exposes a REST service with API methods to run these actions:

- Get events or messages by filter
- Get facets response
- Retrieve document payload based on messageId
- Stream document payload based on messageId
- Ping call for oneviewapi

The ION OneView API is located in the Infor ION (IONSERVICES) API suite and the `/oneviewapi` endpoint. For detailed information about the methods that are exposed in the endpoint, see the Swagger documentation. For information on how to use ION APIs and how to interact with Swagger documentation for the API methods, see the *Infor ION API Administration Guide*.

You should be authorized as administrator to use an ION service API such as ION OneView API.

The methods in this endpoint are authorized at the user level. Internal verification is performed to determine whether the user who calls this API has the permission to perform this call.

Available API Methods

This table shows the API methods available in the `/oneviewapi` endpoint:

Method	Type	Description

`/data`	GET	Get events or messages based on the filter. Define the 'store' parameter to access the type of collection to address: Event (default) or Message. For example, use this query:
		`/data?[filter=<filterExpression>]&[store=<Event|Message>]&[page=<pageNumber>]&[records=<numberOfRecords>]&[sort=<sortFieldName>]`
		You can use the 'filter' URL parameter to express "dynamic" search query by the client.
		Example of compound and nested condition:
		`(event_source_type eq '*jdbc') and (event_type eq 3 OR (event_type eq 4 AND event_source_name eq "infor.rec.rec") )`

`/data/facets`	GET	Get aggregations for statistics analysis. For example, you can find the mostly used connection points. Parameters can be one of the available fields, such as message header and content fields, document type, document id, or document noun. The fields slightly differ for on-premises and in cloud; see the *Infor ION Desk User Guide*.
`/data/documentPayload`	GET	Retrieve a document payload for the specified message Id.
`/data/streamDocumentPayload`	GET	Stream a document payload for the specified message Id.
`/data/ping`	GET	Test whether the service is running.

ION Process API

The ION Process APIs expose functionality that is related to:

- Workflows that run in the Workflow engine.
- Alerts, tasks, and notifications managed by the Pulse engine.

The ION Process APIs are located in the “Infor ION” API Suite and these endpoints:

- `process/application`

  This endpoint exposes methods that are authorized at application level and do not require a user identification. After a client application has access to this endpoint, it can call any method without further security checks being applied.

- `process/user`

  This endpoint exposes methods that are authorized at user level. Internal verification is performed to determine whether the user who calls this API has the permission to perform this call. For example, you cannot close a task that is not assigned to you.

For detailed information about the methods that are exposed in each endpoint, see the Swagger documentation of each endpoint. For more information on how to use ION APIs and how to interact with Swagger documentation for the API methods, see the *Infor ION API Administration Guide*.

Data Catalog API

The Data Catalog is an application that runs in the ION Grid. It exposes a REST Service with API methods to run one of these actions:

- Retrieve a list of all existing objects in the Data Catalog.
- Register object metadata for JSON and DSV objects.
- Retrieve metadata for JSON and DSV objects.
- Retrieve a list of BOD nouns.
- Retrieve noun properties for BODs.

Interface and consumption methods are exposed through the Data Catalog API Service registered within the ION API Suite for Infor ION.

For more information on using ION APIs and interacting with Swagger documentation for the API methods, see *Infor ION API Administration Guide*.

Additionally, the API uses the OAuth 1.0 authorization type. To use this method of authentication, you must obtain security credentials for the DataCatalog service from your system administrator. For a technical description of the API methods, see the swagger documentation of the Data Catalog endpoint on:

`https://<your server name>:9543/datacatalog/swagger.json`

Note: The port number may be different on your installation.

To check whether the `/datacatalog` endpoint is started, run the ping method:

`https://<your server name>:9543/datacatalog/ping`

You can run this method from a browser window. A successful reply returns the REST API version number, such as 1.

Available REST APIs

This table shows the available API methods:

Method	Type	Description
`/datacatalog/ping`	GET	Verify whether the REST Service is running. If successful, the reply contains the API version number.
`/datacatalog/v1/document/list`	GET	Returns a list of all documents in the Data Catalog. Optionally, you can filter by document type or document name. **Note:** This API has been deprecated. Use `/datacatalog/v1/object/list` instead.
`/datacatalog/v1/document/json`	PUT	Upload a document of type JSON with its schema and properties. For validation details, see Defining a custom object of type JSON on page 95. Documents are uniquely identified by their name. A subsequent upload overwrites the previous document definition. **Note:** This API has been deprecated. Use the POST `/datacatalog/v1/object` API instead.
`/datacatalog/v1/document/json/{name}/schema`	GET	Returns the JSON schema for the specified document name. **Note:** This API has been deprecated. Use `/datacatalog/v1/object/{name} /schema` instead.

Method	Type	Description
`/datacatalog/v1/document/json/{name}/properties`	GET	Returns the JSON properties for the specified document name. **Note:** This API has been deprecated. Use `/datacatalog/v1/object/{name} /properties` instead.
`/datacatalog/v1/object`	POST	Upload an object to the Data Catalog. For validation details, see the "Defining a custom document of type …" section for the specific type. **Note:** This API cannot be used to upload a BOD object. Objects are uniquely identified by their name. A subsequent upload overwrites the previous object definition for that type. If an object has the same name as another object of a different type, the import for the new object fails.
`/datacatalog/v1/object/list`	GET	Returns a list of all objects in the Data Catalog. Optionally, you can filter by object name or object type.
`/datacatalog/v1/object/{name}`	GET	Returns the object name, type, subtype, schema, properties, `lastUpdatedOn`, and `lastUpdatedBy` for the specified object name.
`/datacatalog/v1/object/{name}/schema`	GET	Returns the object schema for the specified object name. **Note:** This API cannot be used to retrieve BOD or ANY objects.
`/datacatalog/v1/object/{name}/properties`	GET	Returns the object properties for the specified object name. **Note:** This API cannot be used to retrieve BOD or ANY objects.
`/datacatalog/v1/object/{name}/type`	GET	Returns the object type for the specified object name.

Method	Type	Description
`/datacatalog/v1/object/fetch`	POST	Returns the object name, type, subtype, schema, properties, `lastUpdatedOn`, and `lastUpdatedBy` for each of the objects specified in the API call. Up to 100 object names can be specified in the call. **Note:** This API cannot be used to retrieve BOD metadata
`/datacatalog/v1/object/fetch/audits`	POST	Returns the name, type, subtype, `lastUpdatedOn`, and `lastUpdatedBy` information for each of the objects that are specified in the API call. Up to 100 object names can be specified in the call. **Note:** This API cannot be used to retrieve BOD information.
`/datacatalog/v1/object/summary`	GET	Returns the total object count and the most recent `lastUpdatedOn` value for objects in the Data Catalog. Optionally, you can filter by object type. **Note:** This API excludes BODs

Business Rules API

The Business Rules APIs expose functionality to request the execution of an approval matrix or a decision matrix. These APIs can only execute matrices that have been created and approved in the Business Rules UI in ION Desk. These APIs are located in the “Infor ION” API suite in the “businessrules” endpoint.

Note:

- Only the matrices that are approved can be listed and executed through the Business Rules API.
- The latest version of the matrix is used for each API call.
- To execute a matrix, values are required for the matrix input parameters. The execution of a matrix returns the same results as the Simulation operation in the Business Rules UI. For example, the results of an approval matrix execution could be a list with distributions to the same user, displayed several times. The application that calls the API decides how to handle the approval chain. For example, the application can perform one of these actions:

    - Merge duplicate distributions and send tasks one after the other, as in the workflow task chain in ION.
    - Simultaneously send parallel tasks to all users in the resulting list.
- The result of an approval matrix execution is a list with distributions to users. These users are represented by the **IFS Person ID**, **Group Names**, and **ManagerOf** properties from Infor Ming.le User Management. To retrieve the other relevant properties of these distribution elements, the application that calls the API must be integrated with Infor Ming.le User Management.
- The result of a decision matrix execution is a list of values for the output parameters from the matrix definition. The values from the first row that matched the matrix conditions, which are evaluated for the values provided for the matrix input parameters, are returned. For detailed information about the methods that are exposed in this endpoint, see the Swagger documentation.

For more information on using ION APIs and interacting with Swagger documentation for the API methods, see the *Infor ION API Administration Guide*.

Data Lake API

The Infor Data Lake is a scalable, elastic object store for capturing raw data in its original and native format. The Data Lake provides several interfaces for:

- Retrieving, querying or purging data.
- Providing data management features to manage archiving by policy.
- Restoring aged data.
- Retrieving statistical information about the stored objects.

Interface and consumption methods are exposed through the Data Lake API Service registered within the ION API Suite for Infor ION.

Note: The Data Lake JDBC driver is only available for multitenant Cloud customers using Birst.

For more information on how to use ION APIs and how to interact with Swagger documentation for the API methods, see *Infor ION API Administration Guide*.

For more information about Storage Policies, see *Infor ION Desk User Guide*.

This table shows an overview and statistical information about data objects:

Method	Type	Description
`/v1/status/overview`	GET	Get the total number of data objects and total size.
`/v1/status/dataIngestion`	GET	Get the size and count of the data objects in Infor Data Lake.
`/v1/status/topObjects`	GET	The top 10 biggest Data Objects Names that are stored by their size.
`/v1/status/aggregation`	PUT	Aggregation on one of data object fields such as `dl_id`, `dl_instance_count` or `dl_size` within a defined time range period and interval.

Method	Type	Description
`/v1/status/ping`	GET	Ping call for the Infor Data Lake API.
`/v1/status/version`	GET	Get the current running build number of the Infor Data Lake.

Retrieving data objects

The Data Lake API provides methods to list and retrieve data objects from the Data Lake. You can either retrieve the requested data objects in request using:

- The `streambyfilter` method.
- A two-step approach by first calling the list method and then retrieving the details for each item using the `streambyid` method.

This table shows an overview of the available API methods:

Method	Type	Description
`/v1/payloads/list`	GET	Returns a list of data objects and its indexed metadata including its physical address (Data Object ID) that is based on any defined filter criteria.
`/v1/payloads/streambyfilter`	GET	Returns data object streams as a multipart mixed message that is based on filter arguments passed in the API request. It can be configured to return both corrupt and non-corrupt objects.
`/v1/payloads/streambyid`	GET	Returns a single data object as an octet-stream message that is based on the address of an object stored in Data Lake. Object IDs can be retrieved by passing filter arguments into the `/v1/payloads/list` API and capturing the `dl_id` value from the API response.
`/v1/payloads/markcorrupt`	PUT	Marks an object `dl_corrupt` field to true. This object is not returned with the default `/v1/payloads/streambyfilter` usage.

By default, content in the Data Lake is both stored and streamed to clients in a compressed state. For exceptionally large content retrievals, especially through the `streambyfilter` API, this deflating content method ensures that performance of the gateway and requesting clients remains nominal.

Authorized API applications and `RESTful` API clients that are used for API testing can advertise supported content encoding to the server. To stream and persist data in a compressed format, the requesting party can configure their request with the request header:

- Accept-Encoding: deflate

Using the “identity” value in a request HTTP header, clients can stream their requested content with no encoding in place. This setting is typically configured using the request format:

- Accept-Encoding: identity

Not all clients support the “identity” value. See your API application or client’s documentation to determine whether these request HTTP header values are supported.

Querying data objects

Data Lake API provides a group of methods called Compass.
Note: Data Lake API Compass group of methods is not available in AWS GovCloud

With Compass you can query and retrieve data that is stored in the data objects from the Data Lake.

To run a query and get the results you can use these methods:

- Post a Compass job to receive a unique Query ID for the job.
- Get the status of a job, using the Compass Status method and the Query ID for the job.
- Get the results of a finished job, using the Compass Results method and the Query ID for the job.

For more details on Data Lake Compass and querying data objects see Data Lake queries on page 133.

This table shows an overview of the available API methods:

Method	Type	Description
`/v1/compass/ping`	GET	Tests whether the Compass service is running.
`/v1/compass/jobs`	POST	Use this API to submit a Compass SQL query as an asynchronous job. In the request body parameter provide your SQL statement. The response provides a Query ID. Use the Query ID with the Compass Status API to check the status of the submitted job.

Method	Type	Description
`/v1/compass/jobs/{queryId}/status`	GET	Use this API to check the status of a Query ID that was provided from a Compass job. The response indicates if the query is still processing or has finished. Use the polling parameter to specify the long-polling time-out period. Query statuses: • RUNNING - Indicates that the query is executing. • TRANSFORMING - Indicates that the query results are being prepared. • FINISHED - Indicates that the query completed successfully. You may get the results. • FAILED - Indicates that the query is finished and ended in error. Use the GET result API call to retrieve the error details.
`/v1/compass/jobs/{queryId}/result`	GET	Use this API to retrieve the results for a Query ID that was provided by the Compass Jobs API. Initially, check the status of the Query ID using the Compass Status API. Ensure the process has finished running before using this API to retrieve the result.

Purging data objects

The Data Lake API suite provides methods to purge data objects from Data Lake.

You can purge the data object using one of these methods:

- The `purge/filter` method.
- A two-step approach by first calling the `v1/payloads/list` method to receive the list of data object id(s) and then purging data object(s) using the `purge/ids` method

This table shows an overview of the available API methods for purge data object:

Method	Type	Description
`/v1/purge/ids`	DELETE	Purge data object by its ID. You can retrieve object IDs by passing filter arguments into the `/v1/payloads/list` API and capturing the `dl_id` value from the API response.

Method	Type	Description
`/v1/purge/filter`	DELETE	Purge data objects based on filter arguments passed in the API request.

Note: Purging data objects from Data Lake deletes the data object payload from ION OneView.

Archiving data objects

The Data Lake API provides archive methods to move data objects from frequent access storage in the long-term storage.

After data objects are demoted to long-term storage, they are not accessible to any data retrieval interface. This includes the Data Lake's Payload API methods, Compass API methods, Birst, and other applications. You must restore the data object from archive.

The storage policy is assigned to a data object when the object is ingested and stored in Data Lake. You cannot assign a different storage policy or remove the existing one on data objects that are already stored in Data Lake.

This table shows an overview of the available API methods:

Method	Type	Description
`/v1/archive/strategy`	GET	Get a list of storage policies and data objects assigned to them. By default, all objects are assigned the Default Storage Policy.
`/v1/archive/strategy`	POST	Change an existing or define a new storage policy.
`/v1/archive/logs`	GET	History for created or changed storage policies.

Restoring data objects

The Data Lake API provides methods to restore data objects from the long-term storage to the frequent access storage.

To use the Restore API:

1. Run the `GET /v1/restore/list` method to search for data objects in the long-term storage.
2. Check the response from `GET /v1/restore/list`.

   The response body contains a list of data object addresses that were found in the long-term storage.
3. Use the list of data objects as input in the `POST /v1/restore/payloads` method to start the restore procedure.
4. Run the `GET /v1/restore/logs` method to see the status and history of all restore requests.

   This table shows the API methods:

Method	Type	Description
`/v1/restore/list`	GET	Searches for data objects in long-term storage and returns a list of found data objects that is used an input in `POST /v1/restore/payloads`.
`/v1/restore/payloads`	POST	Starts the restore procedure. The list of data objects from `GET /v1/restore/list` is used.
`/v1/restore/logs`	GET	List of all restore procedures and their status.

Note: Permanently restored data objects cannot be moved back to the long-term storage.

It can take up to 6 hours until the restore is finished and having the data objects available in frequent access storage.

Chapter 15: Data Lake queries

With the Data Lake query functionality, you can query Data Lake data objects.

The functionality is useful to run ad-hoc queries and to execute queries and send results to downstream applications.

Two query methods are available:

- A Data Lake JDBC driver for Birst
- Compass queries

The query functionality and query syntax are different for each method. The functionality is outlined in separate sections.

Method one, the query functionality for Birst, uses a Data Lake JDBC driver to access Data Lake data. The query functionality is only available through a Birst agent.

See Data Lake JDBC driver for Birst on page 136.

Method two, the query functionality for Compass, is available through the Data Lake Compass query editor, the Compass APIs, and the Compass JDBC driver.

For the Data Lake Compass query editor, see "Compass UI" in the *Infor ION Desk User Guide*.

For Compass, see Data Lake Compass queries on page 143.

Note: Compass queries are not available in AWS GovCloud.

Data Lake data object definitions

The Data Lake objects available to query through either method are new-line delimited JSON objects and delimited-separator value (DSV) objects. The required formats are outlined.

JSON data objects

For data objects in JSON format, the Data Lake JDBC driver for Birst and Compass supports newline-delimited (streaming) JSON format.

Newline-delimited JSON format offers better query performance and is a recommended format for big data applications. The JSON objects must have a flat, single-level structure. The only nested structure supported is for localized strings. JSON objects must use UTF-8 encoding.

See Creating custom metadata on page 92.

DSV data objects

The Data Lake JDBC driver for Birst and Compass supports queries of Data Lake objects in DSV (delimiter-separated values) format.

For data objects in DSV format, the DSV payload objects must include a single header row with property names. If DSV metadata dialect “skipLines” is not explicitly defined in the metadata, a value of one is assumed. If DSV metadata dialect “headerLine” is not explicitly defined in the metadata, a value of one is assumed. Each DSV line must end with a carriage return or line feed character, or a carriage return and line feed character.

The DSV object must use a single separator character. The DSV metadata dialect “separator” value defines the separator character. The separator characters of comma, tab, and pipe are supported.

The DSV metadata dialect `enclosingCharacter` property is a character that is used to define the start and end of a value. Each column and value does not require enclosing characters. If a value in a data object contains the specified enclosing character, it must be escaped. You can add another enclosing character directly next to it. When a pair of enclosing characters are found inside a field, they are interpreted as a single character. The enclosing character of single or double quotes is supported.

Special characters in values can produce inaccurate or incorrect results. If a query returns inaccurate results, verify that the value is enclosed. DSV objects must use UTF-8 encoding.

See Creating custom metadata on page 92.

Data object definitions for localized string values

Data Lake objects may contain localized string values.

The values can be queried by the Data Lake JDBC driver for Birst and by Compass queries.

The localization functionality works in conjunction with the Data Catalog Locale Selection function. The Locale Selection function is used to define the locales that supported the source application. Or the function is used for defining the locale codes for the localized values in the Data Lake “payload” objects.

The term “payload” denotes the actual DSV and JSON objects that are stored in the Data Lake. The source application sends data objects, with the localized values, to Data Lake. The query matches the locale selections to the localized strings in the Data Lake data. The query returns localized values into numbered position columns. The positions, and the main and substitute locale codes for each position, are defined in the Locale Selection.

For Data Catalog Locale Selections, see the *Infor ION Desk User Guide*.

For example, the source application supports these locales:

- `en_US`
- `fr_CA`
- `es_ES`

The Data Catalog locale selections are position 1 for `en_US`, position 2 for `fr_CA` and position 3 for `es_ES`.

The locale in the first position is used as the default locale. When a query selects a localized value, the driver retrieves the `en_US` value for table column 1. The `fr_CA` value is retrieved for table column 2 and the `es_ES` value for table column 3. The column name does not include the locale code. With the numbered position columns, you can specify which locales are used.

Compass queries and the JDBC driver for Birst support two string localization methods. Both methods use the data object's property metadata, stored in Data Catalog Object Schemas.

The query reads the source Data Lake objects for a match to the Data Catalog's Locale Selection "Locale Code Search List". The search list must include the locale code first and can be followed by one or more locale codes to use as substitute locales. For example, if a value for `en_US` is not found in the data, and the next code in the list is "`en`". The driver returns the "`en`" value.

The matching process between the query, the Locale Selections and Data Lake data occurs each time the query runs. The data object localization definitions are explained later.

For query processing using the JDBC driver for Birst, see these sections:

- Data Lake JDBC driver for Birst on page 136
- Data Selection for Localized String Values on page 137

For query processing using Compass queries, see these sections:

- Data Lake Compass queries on page 143
- Queries for localized data on page 157

Data object localized data definition, method 1

In this method, the data object's metadata defines the property as type object and localized.

In the JSON data object payload, the localized string values are formatted as an object. Each localized value is a name value pair using the locale code and the localized string value. The driver reads the JSON data object to match the locale codes to the localized values.

Example of a localized property in a JSON data object:

```
"greeting":{"en_US":"hello","fr_FR":"bonjour","es_MX":"hola"}
```

Data object localized data definition, method 2

In this method, the data object's metadata defines the property as type string, and the property is defined as a localized property.

In the JSON or DSV data object, the localized string data is formatted as individual properties. Each property name has a locale code suffix. The driver reads the JSON or DSV data object to match the property suffixes to the locale codes, and returns the localized values.

Example of a JSON data object with strings that are localized:

```
“greeting_en_US":”hello”,”greeting_fr_FR”:”bonjour”,”greeting_es_MX”:”hola”
```

Data Lake JDBC driver for Birst

The Infor Data Lake JDBC driver for Birst Analytics is used to query JSON and DSV objects stored in the Data Lake.

The driver provides a method to retrieve data from data objects sent through ION Data Lake flows. The metadata for the objects is stored in the Data Catalog as object schemas.

The JDBC driver is used with the Birst Connect agent and is associated with Birst connections. The JDBC driver is used to select and retrieve data objects from the Data Lake and returns the data for Birst queries.

Data Selection Features

Birst Connect uses the JDBC driver to select data objects from Data Lake. You can use Query-based objects to input a query to retrieve data, sourced from JSON or DSV payload objects, as rows of a result set.

Note that in Birst queries for DSV data objects, omit lines with nothing but whitespace and null values. For example, a line of separator characters between empty values results in an empty row of data when the object is queried. DSV payloads with blank lines, such as lines with only line feed or carriage return characters, are not supported. The data object fails because the row does not contain the required number of fields.

Two query methods are supported:

- A general object query is a select statement to return all properties and data from an object. The Birst query is not visible. A general object query returns the data in the latest object loaded into Data Lake.
- A query-based object selection is a select statement to return all or specific properties from an object. If a WHERE clause condition does not base the query on the `lastModified` timestamp, the driver returns data from the latest data object in the Data Lake. If a WHERE clause condition bases the query on a `lastModified` timestamp, the query returns data from objects loaded within the `lastModified` limits. The `lastModified` value corresponds to the datetime at which the data object was loaded into Data Lake.

Through the Birst connection, use the **Edit objects** option to display a list of the objects available in Data Lake. The list is comprised of JSON or DSV type object types defined in the Data Catalog.

The driver uses the Data Catalog metadata for the object to identify object names as table names and the object properties as table column names.

Data Lake objects marked as corrupt are automatically excluded from query results.

Data Selection for Localized String Values

The localization functionality supports queries for localized string values, stored in the data objects in Data Lake.

Through the object metadata, specify that a property is localized, and the driver returns the localized values. Localized values are stored in data stores, where they can be used in reports and dashboards.

For details on defining localized string data and metadata, see Data object definitions for localized string values on page 134.

Specifically, for Birst, the object query includes the locale keyword to select localized string values. Additionally, a separate table in the Birst data store stores the Data Catalog locale selections. For example, the source application supports these locales:

- `en_US`
- `fr_CA`
- `es_ES`

The Data Catalog locale selections are position 1 for `en_US`, position 2 for `fr_CA`, and position 3 for `es_ES`.

The locale in the first position is used as the default locale. When a query selects a localized value, the driver retrieves the `en_US` value for table column 1. The `fr_CA` value is retrieved for table column 2 and the `es_ES` value for table column 3. The column name does not include the locale code. With the numbered position columns, you can specify which locales are used in Birst.

To complete the localization for Birst, the locale selections are setup as a separate table in the data store. The locale selections are used to match the table data to the report consumer's language. When a report is rendered, if the report consumer's language is `en_US`, the report displays the data in position 1. If the report language is fr_FR, the report displays data in position 2. If the report language is not available in a position, the report returns the value in position 1, because it is the default locale.

The driver uses two methods to determine if string properties are localized. Both methods use the document's property's metadata, which is stored in Data Catalog Object Schemas.

The driver reads the source objects for a match to the Data Catalog's Locale Selection "Locale Code Search List". The search list must include the locale code first, and can be followed by one or more locale codes to use as substitute locales. For example, if a value for `en_US` is not found in the data, and the next code in the list is "`en`", the driver returns the "`en`" value.

The matching process between the driver, the Locale Selections and Data Lake data occurs each time the query runs.

If the Data Catalog Locale Selections change after data is loaded into the target tables, the target tables can be cleared and repopulated. For example, if position 2 is French, French values are loaded into

the target tables. If position 2 changes to Spanish, subsequent queries populate position 2 with Spanish values. A report run for the Spanish language shows Spanish for current data and French for historic data. The report shows a combination of both, since the current locale selection defines position 2 as Spanish.

SQL Query Expressions

Each SQL query is based on a single JSON or DSV custom document. Joins are not supported.

The object names are not case-sensitive. For example, an object schema named Customer may be queried as Customer or customer.

Property names are case-sensitive. For example, a property name of ProductID must be selected in a query as “select ProductID”.

Use double-quotes if the object name or property name includes spaces. Double-quotes are not required if the object or property name includes underscores. Object or property names that begin with a digit must be enclosed within double-quotes. Enclose property names that are SQL reserved words with double-quotes.

Select individual properties from a data object

Example: `SELECT property1, property2 from dataobject`

Select individual properties from the data object. Object names are not case-sensitive. Property names are case-sensitive. In the example that is mentioned earlier, “property1” and “property2” are property names. The case that is used in the query must match the case of the property names in the Data Catalog’s metadata definition.

The result of a query without a WHERE `lastModified` condition are the "rows" of data in a single payload object. The rows correspond to the tuples in the latest object loaded into the Data Lake. In BIRST the properties are returned in alphabetical order, regardless of the order in which they are specified in the select statement.

Select individual properties using aliases from a data object

Example: `SELECT property1 as alias1, property2 as alias2 from dataobject`

Select individual properties from the data object and use aliases. Aliases can be enclosed in square brackets or double quotes.

Incorporate comments in a query

Example: `/*query that uses aliases*/ SELECT property1 as alias1, property2 as alias2 from dataobject`

Enclose comments within `/*` and `*/` characters.

Select properties and include the Data Object load timestamp

Example: `SELECT property1, property2, lastModified from dataobject`

Select the `lastModified` column in the query to return the last modified datetime of the data object. The last modified datetime is the timestamp, in UTC, in which the message containing the “row” was posted to the Data Lake. `lastModified` is case-sensitive. The last modified timestamp is a property of the data object itself. It is not a property inside of the object payload.

Select records that are based on the last modified timestamp

Example: `SELECT ... from dataobject where lastModified >= 'Timestamp in UTC'`

Select records using the last modified datetime to limit the selection to data posted to the Data Lake within a specific time frame. The `lastModified` value represents the timestamp, in UTC, for the datetime in which the data object was added to the Data Lake. This syntax is useful because you can append the `lastModified` condition to retrieve data objects loaded into the Data Lake based on the object's load time. This is the main difference between selecting objects using the standard Birst object selection and a query-based selection.

The timestamp is a property of the object itself. It is not a property inside of the object payload. Through Birst, you can setup a variable to define the timestamp value, which can be used in place of specifying a timestamp value. Enclose the timestamp value, or the variable, in single quotes.

The lastModified timestamp format is a timestamp in UTC. The timestamp format is:

`yyyy-MM-dd'T'HH:mm:ss.SSS'Z'`

A last modified reference in a query expression’s WHERE clause can use these operators:

- =
- >=
- <=
- `BETWEEN`

Select all records

Example: `SELECT * FROM dataobject`

The select all syntax is not recommended. The select all syntax allows the Birst query to retrieve all properties from the object. This functionality is useful when the object properties are updated over time. If the data object definition changes over time, it may not match the Birst table schema into which the data is imported.

Select the Data Catalog Locale Selections

Example: `SELECT * from datacatalog.locale_selection`

The query returns the locale selections that are defined in the Data Catalog Locale Selection function. The CODE is the locale code. The POSITION is the position number of the locale. The SEARCH_LIST values are the locales used by the query to select localized values. The search list locales are searched in priority order until a value is found. The DEFAULTING value combines two properties. The Default value designates the locale code of the default locale. The default locale is automatically assigned to

the locale in position 1. The `NoFallback` value denotes that if a localized value is not found in the data, the position is empty. The `NoFallback` option is automatically assigned.

Select localized values for a property

Example: `SELECT locale(localizedproperty, number of positions) FROM dataobject`

The locale function is used to select localized values from the source object. The locale function is case-sensitive, and all letters are lower-case. The number of positions defines the number of numbered columns that are returned in the results. The locale function returns localized values that are based on the locale selections.

The query results are the property name with a position number suffix, such as property_1, property_2, property_3. The values that are populated in the positions are the localized values matching the locale code, or a search list locale code, for each position.

If the data object does not contain localized values, the property's string value is returned in position 1. The remaining position columns are empty.

Select localized values for a property that is not localized

Example: `SELECT locale(property, number of positions) FROM dataobject`

The locale function is used to select localized values from the source object. The number of positions defines the total number of numbered columns returned in the results. The property is not defined as a localized property in the metadata. The result is the position number columns. The single string value found in the data object, is returned as the value of position_1. The remaining position columns are empty.

Select a property that is localized without the locale function

Example: `SELECT localizedproperty FROM dataobject`

When the locale function is not used, a single value, in the default locale, is returned. The default locale is the locale in position 1 of the Locale Selections. If the data does not contain a localized value for the default locale, or the locale's search list locale codes, the value is empty.

Troubleshooting SQL expressions

The Troubleshooting information describes how to diagnose and solve issues with SQL expressions.

The query returns no values or displays the message: "There are no columns to display. Please adjust your search criteria or applied filters".

Verify in OneView that at least one data object of this type is received. It must contain data with property names matching those in the Data Catalog and your query.

For DSV objects, verify that the object payload has a single row of column headers. The column headers must match the metadata property names. Property names are case-sensitive.

If the query contains the locale function, verify that the locale keyword is lower-case. The maximum number of locale positions is 32.

The query returns the error: Unable to retrieve data…Invalid property name reference: 'property name'

The property name(s) reference the property names that are used in the query. Verify that property name(s) exactly match the property names defined for the object in the Data Catalog. Property names are case-sensitive.

For DSV objects, verify that the object payload has a single row of column headers. The column headers must match the metadata property names. Property names are case-sensitive.

The query returns the error: Object 'object name' not found.

The object refers to the table name that is referenced in the query. Verify that the object name is correct, and verify that the object name matches the name that is defined in the Data Catalog. Object names are not case-sensitive.

The lastModified predicate in the WHERE clause is not working.

The keyword `lastModified` is case-sensitive with an upper-case `M` for Modified. The timestamp must be formatted using the ISO-8601 timestamp format. Timestamps must be provided using UTC timezone conventions and include a fractional millisecond. The timestamp format is: `yyyy-MM-dd'T'HH:mm:ss.SSS'Z'`

Example: `2018-03-17T09:30:00.001Z`

The lastModified in the WHERE clause returns an error 500

The timestamp value must contain milliseconds.

At least two columns are required to configure 'dataobject query' object

This error is displayed when the query runs.

In Birst, the `SELECT` statement must query a minimum of two properties.

Exception occurred when executing SQL query: Property predicate is not supported.

The `SELECT` statement cannot include a property in the WHERE clause.

Invalid UTF-8 start byte 0x9b" is displayed during Birst data import.

A data object can be used in a query and return the correct results. The import into Birst fails with error: "Invalid UTF-8 start byte 0x9b." The error occurs because the source data objects are not in UTF-8 format. The solution is to format the source objects as UTF-8 and send them to the Data Lake again.

To omit the Data Lake object from a query, mark the object as corrupt. Corrupt objects are automatically excluded from queries. See Retrieving Data Objects for information to mark Data Lake objects as corrupt.

Alternatively, use the `lastModified` property to avoid the invalid object.

A SQL parsing error is displayed when retrieving data and object and property names contain special characters, such as hyphens or slashes.

The property names with special characters, such as hyphen (-) and slash characters (/,\), must be enclosed within double-quotes.

Query fails if the lastModified predicate uses > or < lastModified. Replace with >= or <=.

A query fails if the Data Lake has over 10,000 data objects to include in the query results. Update the query predicate to use `WHERE lastModified >= timestamp value` or `WHERE lastModified <= timestamp value`.

The query returns null values for integer, big integer or decimal data types.

The query results are returned in the data type defined in the object's metadata, stored in the Data Catalog. For example, if the metadata property is defined as an integer data type, the data type returned is integer. If the data object contains a different data type, such as a decimal value, the query returns a null value. If a null is returned, verify that the data object's value matches the metadata data type. If not, an option is to update the metadata definition in the Data Catalog to suit the payload values.

Unable to retrieve data for a request.

A data object that is selected by the query is invalid because the object is incomplete or invalid.

For JSON objects, ensure that each JSON tuple in the data object is complete.

For DSV objects used by the JDBC driver for Birst, ensure that the payload object does not contain extra rows with only carriage return, line feed or carriage return and linefeed characters. An error is displayed that the line contains a different number of fields than the header.

The message includes the Data Object ID, which is the unique identifier for a Data Lake object. Exclude the data object from the query or mark the data object as corrupt. Data objects marked as corrupt are automatically excluded from the results. Use the Data Object ID value to retrieve and review the data object through Infor OS.

The locale function returned localized values, but the values are unexpected.

Verify that the Data Catalog Locale Selections are populated. The driver uses a default locale list when no locale selections are available in the Data Catalog. You can select the Data Catalog locale selections and use the Data Catalog Locale Selections function to add or update locale selections.

See the "Locale Selections" in the *Infor ION Desk User Guide*.

The query is hanging. No errors and no data are returned.

Verify that the environment running Birst Connect 2.0 agent has Java version 1.8 or later.

Data Lake Compass queries

Data Lake Compass query functionality provides robust query capabilities to search and extract results from data stored in the Data Lake.

Query functionality is available through the **ION Desk > Data Lake > Compass** function, the Infor OS `datalakeapi` Compass endpoint, and the Compass JDBC driver.

The **Compass** menu option provides a query editor through a user interface. See "Compass UI" in the *Infor ION Desk User Guide*.

The Compass APIs provide methods to submit queries, poll for status, and retrieve results.

See Querying data objects on page 129.

The Compass JDBC driver may be set up with an SQL query tool. You can run queries through a local SQL query tool.

Using Data Lake query, you can query new-line-delimited JSON objects and DSV objects that are stored in the Data Lake. You can combine the JSON and DSV object data in queries using joins.

Query functionality and syntax

In general, the query language follows Microsoft SQL Server syntax. The purpose of this section is to explain general query syntax, supported functionality and expressions for Data Lake Compass queries.

The documentation outlines special considerations for Data Lake queries. This documentation is not intended as comprehensive SQL documentation.

Specific functionality for Apache Spark is also supported.

Compass queries may also use Infor

The Compass query syntax is the same regardless of the query method. The same query may be run in the functions, keywords and administrative stored procedures.Compass ION Desk function, through the Compass APIs, and through the Compass JDBC driver. The differences in the execution and output formats are noted.

A prerequisite for all queries is Data Catalog metadata for data objects. You may query one or more data objects in a query, and you may combine data object types in the same query. For example, a query may select data from JSON objects joined to DSV objects.

Note: In the Data Catalog, object names are not case-sensitive. In the Data Catalog metadata, properties are case-sensitive. For Compass queries, all object names and property names are case-insensitive. Therefore, do not use duplicate property names in a data object with different capitalization. For example,

the Compass query functionality cannot detect a difference between properties named ITEM, Item and item. Use distinct property names as a rule.

These characters are supported for object and property names:

- A through Z in upper or lower case
- 0 through 9
- Underscore
- Space

Use double-quotes around object names and property names that include an underscore or spaces, or begin with a number. Use double-quotes around names that are SQL reserved words.

Query syntax is case-insensitive. The documentation uses uppercase to highlight the query functionality and keywords. Uppercase is not required unless specifically noted for a function or keyword.

Data values in Data Lake data objects are case-sensitive. Values such as “ABC” and “abc” are distinct.

The expression syntax and special considerations are outlined for each function or expression. Note:

- Compass queries support SELECT statements and administrative EXEC stored procedures.
- Compass queries do not support CREATE, INSERT, UPDATE, DELETE, or DROP.
- Compass functions, keywords and administrative stored queries do not support the use of `;` to execute multiple queries.

Note: Query constructs, keywords, and functions that are not documented may work. But the query results may not match the expected results and can change in a future release.

Query structure

SELECT

Use SELECT to select values from data objects stored in the Data Lake. Each value that is specified in a SELECT is either:

- A property defined in the metadata for the data object.
- A literal value.
- A calculated value or expression.

The results are represented in rows and columns.

Data Lake is not a traditional transactional database. Data Lake stores every variation, or version, of a record. Historic data is available, and historic versions of data is available. By default, a query retrieves the highest variation of a record that is not tagged as deleted.

See Variation handling on page 166.

In the examples that are provided, `dataobject` represents the Data Catalog object, like a database table. Property represents the Data Catalog object property, like a table column.

The query structure follows Microsoft SQL Server.

Syntax

```
SELECT one or more properties, expressions or literals
FROM dataobject
WHERE filter condition(s)
```

```
GROUP BY expression
HAVING filter condition(s)
ORDER BY expression [ ASC | DESC ]
```

FROM

Use a FROM clause to specify the data objects or subquery from which the selected values are stored or derived. Compass queries may include data from more than one data object. See query functionality for JOIN and Subqueries for information.

WHERE

Use a WHERE clause to specify filter conditions on which to base the results. Most Compass queries should include a WHERE clause to limit the query results.

Queries executed through the Infor ION `datalakeapi` Compass API endpoint do not limit query results. Therefore, a query, by default, returns all data.

The Data Lake may be used as a source of data to populate a data warehouse or a data store. It is common to use initial load and incremental load logic. The initial load retrieves all data from Data Lake. Subsequent queries retrieve incremental data, or data loaded since the last extract. You can use the lastmodified timestamp on the WHERE clause of queries to retrieve incremental data.

See Queries for incremental data loads on page 158.

Note: Queries executed through the **ION Desk > Data Lake > Compass** function limit results to 100 rows.

GROUP BY

Use a GROUP BY to specify properties or expressions on which to group the results. If the group by contains an expression, the expression must be in the SELECT clause as well. GROUP BY does not support aggregation or windows expressions. GROUP BY references must be property names; alias cannot be used.

HAVING

Use HAVING to specify conditions to filter the results. HAVING is commonly used with a SELECT statement for an aggregated result set.

ORDER BY

Use ORDER BY to sort the result set columns. Use ASC or DESC to specify ascending or descending order. Property names and aliases may be used. In Compass data storage, string values are case-sensitive, which affects the result order.

UNION | UNION ALL

Use a UNION to union data objects into a common result set. Each selection in the union must have the same number of selected properties, and the properties must have the same data types. Use a UNION ALL to retrieve all values, including duplicates.

Syntax

```
Select * from dataobject1
UNION
Select * from dataobject2
```

EXCEPT

Use EXCEPT to return the results of the first query that are not included in the results of the second query. Each selection must have the same data types.

Syntax

```
Select * from dataobject1
EXCEPT
Select * from dataobject2
```

INTERSECT

Use INTERSECT to return only the rows that are in the results of both the first and second queries. Each selection must have the same data types.

Syntax

```
Select * from dataobject1
INTERSECT
Select * from dataobject2
```

WITH

A WITH clause, or common table expression, can be placed above the SELECT clause to use a common table expression. The common table selected values can be used throughout the overall query SELECT clauses. Note that with common table expressions,

Syntax

```
WITH commontablename AS
(SELECT property1 from dataobject1)
SELECT property1 FROM commontablename
```

SELECT ALL

Use select * to retrieve all values from the Data Objects referenced in the FROM clause of the query or subquery.

Example: `SELECT * from dataobject`

SELECT TOP number of rows

Use the TOP clause to select the number of records from a result set. Use an integer to specify the number of rows in the result set. TOP PERCENT and TOP WITH TIES are not supported.

Syntax

```
SELECT TOP number_of_rows * from dataobject
```

SELECT DISTINCT

Use DISTINCT to remove duplicate values from the result set. Use a property value to specify the property in the data object on which duplicate values are based.

Syntax

```
SELECT DISTINCT property1 from dataobject
```

Note that the values in Data Lake are case-sensitive. The same values can be displayed in the results, even though the only difference is the capitalization. For example, values ‘ABC’ and ‘abc’ are different and both values are returned in a select distinct query. To perform a select distinct on case-insensitive values, use upper or lower functions to convert values to upper- or lower-case characters.

SUBQUERIES

Subqueries are used to incorporate more than one SELECT in a query. The second, or subsequent SELECT statements, may be in the main SELECT, the FROM or the WHERE and HAVING clauses.

A subquery in the FROM clause is referred to as an inline view. Compass queries do not support all alias functionality for inline views. For example, a property alias in an inline view cannot have a second alias in the outer query.

Correlated subqueries are not supported.

JOINS

Joins provide a method to query data from more than one data object. The joined objects, or tables, are joined based on a relationship defined in the query. Compass queries supports joins for different object formats. You can define a query joining data from JSON objects and DSV objects.

Syntax

```
select dataobject1.property1, dataobject1.property2, dataobject2.property1, dataobject2.property2, dataobject2.property3 from dataobject1 [ INNER JOIN | OUTER JOIN | JOIN | LEFT INNER JOIN | LEFT OUTER JOIN | RIGHT JOIN | CROSS JOIN] dataobject2 on dataobject1.property1=dataobject2.property1
```

CASE

Case expressions are used to evaluate a list of conditions and return one of multiple values.

Syntax

```
CASE condition_expression
     WHEN expression_value THEN result_expression
     WHEN expression_value THEN result_expression
     ELSE result_expression
END
```

String functions

The following string functions are supported.

CONCAT

Use CONCAT to concatenate one or more values into a string. Use single quotes around string literals. Use double-quotes to enclose property names that begin with a digit or have spaces or other special characters.

Syntax

```
CONCAT(property1,property1,expression) as concatenatedstring
```

LEFT

Use LEFT to return the left characters of a string expression, string literal or property value. The second parameter, `integer_value`, is an integer that defines the number of leftmost characters to return.

Syntax

```
LEFT(stringproperty,integer_value)
```

RIGHT

Use RIGHT to return the right characters of a string expression, string literal or property value. The second parameter, `integer_value`, is an integer that defines the number of characters to the right to return.

Syntax

```
RIGHT(stringproperty,integer_value)
```

LOWER

Use the LOWER function to convert a character string to lowercase. This function is useful because values in Compass data storage are case-sensitive, and LOWER and UPPER functions may be used to convert characters to the same case for result or comparison purposes.

Syntax

```
LOWER(stringproperty)
```

UPPER

Use the UPPER function to convert a character string to uppercase. This function is useful because values in Compass data storage are case-sensitive, and LOWER and UPPER functions may be used to convert characters to the same case for result or comparison purposes.

Syntax

```
UPPER(stringproperty)
```

LTRIM

Use LTRIM to trim, or eliminate, the leading spaces of a character string.

Syntax

```
LTRIM(stringproperty)
```

RTRIM

Use RTRIM to trim, or eliminate, the trailing spaces of a character string.

Syntax

```
RTRIM(stringproperty)
```

TRIM

Use TRIM to trim, or eliminate, leading and trailing spaces from a character string.

Syntax

```
TRIM(stringproperty)
```

REPLACE

Use REPLACE to substitute a string for another. The first parameter is the string expression. The second parameter is the original string to search. The third parameter is the replacement string.

Syntax

```
REPLACE(string_expression, search_string, substitute_string)
```

SUBSTRING

Use SUBSTRING to return a substring, or portion, of a string expression. The first parameter is the string expression. The second parameter is the starting position. The third parameter is the number of characters in the result. If the number of characters is negative, the result is an empty string.

Syntax

```
SUBSTRING(string_expression, start_position, number_of_characters)
```

CHAR

Use CHAR to return the ASCII character. The parameter is an ASCII character code. Compass queries supports a range higher than the 255 ASCII codes.

Syntax

```
CHAR(ascii_code)
```

Mathematical operators

These mathematical operators are supported:

- Addition using the + character
- Subtraction using the - character
- Multiplication using the * character
- Division using the / character
- Percentage using the % character

You may also use + to concatenate values, such as strings.

Numeric functions

The following numeric functions are supported.

ISNUMERIC

Use ISNUMERIC to determine if a value or expression is numeric. If the expression is numeric, the result is 1. If the result is not numeric, the value is 0.

Syntax

```
ISNUMERIC(expression)
```

CEILING

Use the CEILING function to return the smallest integer equal to or higher than the number value.

Syntax

```
CEILING(number_value)
```

FLOOR

Use the FLOOR function to return the highest integer equal to or less than the number value.

Syntax

```
FLOOR(number_value)
```

ROUND

Use the ROUND function to round a numeric value to the specified number of decimal places.

Syntax

```
ROUND(number_value, number_of_decimal_places)
```

Signed expression

Use a signed expression to convert a value or numeric expression from negative to positive or positive to negative.

Syntax

```
SELECT -(numeric_expression)
```

Aggregation functions

The following aggregate functions are supported.

SUM

Use SUM to add numeric values and return a single value.

Syntax

```
SUM(numeric_expression)
```

AVG

Use AVG to average the results of numeric values in a result.

Syntax

```
AVG(numeric_expression)
```

COUNT

Use COUNT to count the number of records returned in a query. COUNT may also be used with COUNT DISTINCT to count the number of unique rows.

Syntax

```
COUNT(expression)
COUNT(DISTINCT expression)
```

MAX

Use MAX to return the highest, or maximum value, from a set of values. MAX does not support Boolean properties.

Syntax

```
MAX(expression)
```

MIN

Use MIN to return the lowest, or minimum value, from a set of values. MAX does not support Boolean properties.

Syntax

```
MIN(expression)
```

Datetime functions

The following datetime functions are supported.

CURRENT_TIMESTAMP

Use the CURRENT_TIMESTAMP function to return the current datetime. The timestamp value returned is in UTC in ISO8601 RFC3339 format.

Syntax

```
select current_timestamp
```

Result example

```
2019-09-01T09:30:00.000Z
```

DATEADD

Use the DATEADD function to add or subtract date units for a date. The result is a date or timestamp. The first parameter is the datepart, such as yy for year or month. The second parameter is the number of units to add. The number may be positive or negative. The third parameter is the date or timestamp value or expression.

Syntax

```
DATEADD(datepart,number_of_units,date_expression)
```

DATEDIFF

Use the DATEDIFF function to return the number of date part units between two dates or timestamps, such as the number of days between two dates or the number of hours between two timestamps. The result is a number. The first parameter is the datepart, such as yy for year or month. The second and third parameters are the date or timestamp values to compare.

Syntax

```
DATEDIFF(datepart,date_expression,date_expression)
```

DATEPART

Use the DATEPART function to return the value of a specific part of a date or timestamp. For example, return the hour of a timestamp or the month of a date. The first parameter is the datepart, such as yy for year or mm for month. The second parameter is the date or timestamp expression.

These date part formats are not supported: %D, %U, %u, %V, %w, and %X.

Syntax

```
DATEPART(datepart,date_expression)
```

GETDATE

Use the GETDATE function to return the current date and time. The timestamp is in UTC in ISO8601 RFC3339.

Syntax

```
select GETDATE()
```

GETUTCDATE

Use the GETDATE function to return the current date and time. The timestamp is in UTC in ISO8601 RFC3339.

Syntax

```
select GETUTCDATE()
```

Logical and comparison operators

The following operators are supported.

- AND
- BETWEEN
- IS NULL
- IS NOT NULL
- LIKE
- NOT LIKE
- OR
- GREATER THAN using the > character
- GREATER THAN OR EQUAL TO using the >= characters
- LESS THAN using the < character
- LESS THAN OR EQUAL TO using the <= characters
- ANY
- ALL
- SOME

Do not use the syntax of single quotes with or without spaces to evaluate NULL and NOT NULL conditions. The following query syntax will produce unreliable results:

```
=''
=' '
<>''
<>' '
```

Wildcard characters using % and _ are supported. Wildcard characters using [ ] and [^] are not supported.

Compass data values are case-sensitive. Values ‘abc’ and ‘ABC’ are not equal.

Compass queries, in general, compare strings to integers and vice versa, for comparisons such as `where order_number = 123 or order_number= '123'`.

Conversion functions

The following conversion functions are supported:

CAST

Use CAST to cast a value or expression to a data type. The first parameter is the expression or value. The second parameter is the data type for the result. If the value cannot be cast to the requested type, the result is NULL.

Syntax

```
CAST(expression AS data_type)
```

CONVERT

Use CONVERT to convert a value or expression to a data type.

CONVERT datetime to string

Use the CONVERT function to convert a string to a specific datetime style. The first parameter is the string datatype and size, such as varchar(20), the second parameter is a datetime expression, the third parameter is the date_style, or the style number. Style numbers are defined by Microsoft SQL Server. Date expressions are not supported. Date time styles 130 and 131 are not supported.

Syntax

```
CONVERT(string_type, datetime_expression, date_style)
```

CONVERT string to datetime

Use the CONVERT function to convert a date or datetime to a string. The first parameter is the date or datetime type. The second parameter is the string expression. Date expressions are not supported.

Syntax

```
CONVERT(datetime_expression,string_expression)
```

Analytic functions

These analytic functions are supported:

ROW_NUMBER … OVER … PARTITION BY

Use the ROW_NUMBER function to number the result set. If a partition is used, the row number of the row within the partition is returned. Use ORDER BY to sort the results.

Syntax

```
ROW_NUMBER ( ) over ( [ PARTITION BY value_expression1, value_expression2,
… ] order by …)
```

RANK …OVER … PARTITION BY

Use the RANK function to rank, or number, the row results. If a partition is used the rank is the number of the row within the partition. RANK is used WITH TIES to assign the same rank value to tied results.

Syntax

```
RANK ( ) over ( [ PARTITION BY value_expression1, value_expression2, … ]
order by …)
```

LEAD

Use the LEAD function to access data from the following or a subsequent row of a result set. The first parameter is the expression to select a value from a subsequent row. The second parameter is the offset, such as 1, to indicate the following row. The third parameter is the default, or substitute, value to use if the subsequent row reference is not available.

Syntax

```
LEAD (expression, offset_value, default_value) over (partition_by … order_by …)
```

Spark query functionality

Compass queries support a limited set of Spark query functionality. This section only includes Spark functionality that is not available in SQL Server or does not specifically match SQL Server.

Query syntax

LIMIT

LIMIT is used to limit the result set to a specified number of rows.

Syntax

```
Select * from dataobject LIMIT number_of_rows
```

Datetime functions

Compass queries base all Data Lake timestamp data in the UTC timezone. Queries that use timestamp values are always considered in UTC.

DATE_FORMAT

Use DATE_FORMAT to convert a datetime to a string. The first parameter is the timestamp expression. The second parameter is the format style, for example, ‘YYYY-MM-DD’.

Syntax

```
DATE_FORMAT(timestamp_expression, format)
```

DATE_TRUNC

Use DATE_TRUNC to truncate a timestamp and convert it to the specified datetime unit. The first parameter is the datetime unit. The parameter is enclosed in single quotes. The second parameter is the timestamp value.

Syntax

```
DATE_TRUNC('YYYY',timestamp_expression)
```

CURRENT_TIMESTAMP

Use CURRENT_TIMESTAMP to return the current datetime. The datetime is in UTC.

Syntax

```
Select CURRENT_TIMESTAMP
```

ADD_MONTHS

Use ADD_MONTHS to add or subtract months from a date. The first parameter is the date expression. The second parameter is the number of months to add or subtract from the date.

Syntax

```
select ADD_MONTHS(date_expression, number_of_months)
```

YEAR

Use YEAR to return the year from a date expression.

Syntax

```
select YEAR(date_expression)
```

LAST_DAY

Use LAST_DAY to return the last day of the month associated with a date or datetime.

Syntax

```
select LAST_DAY(date_expression)
```

TRUNC

Use the TRUNC function to truncate a date or timestamp to a year or month. The result is a date.

Syntax

```
INT(trunc(current_date),'MM’)
```

UNIX_TIMESTAMP

Use UNIX_TIMESTAMP to convert a date expression to UNIX timestamp format. The first parameter is a date or timestamp. The second parameter is the format of the date expression, such as ‘yyyy-MM-dd’. The date_format parameter is not required if the date_expression references a date or datetime property in Compass data storage.

Syntax

```
select UNIX_TIMESTAMP(date_expression,'date_format')
```

Comparison functions

NVL

Use NVL to evaluate an expression and return a substitute value if the expression is NULL. The first parameter is the expression. The second parameter is the substitute value.

Syntax

```
select NVL(expression, substitute_value)
```

NVL2

Use NVL2 to evaluate an expression and determine if the value is null or not null. Define a substitute value for a not null condition and a substitute value for a null result. The first parameter is the expression,

the second parameter is the substitute value for a not null condition. The third parameter is the substitute for a null condition. The second and third parameters must be the same data type.

Syntax

```
select NVL2(expression, substitute_value_for_not_null, substitute_value_for_
null)
```

CEIL

Use CEIL to return the smallest integer greater than or equal to the numeric expression value.

Syntax

```
select CEIL(numeric_expression)
```

String functions

INSTR

Use INSTR to return the position of the first occurrence of substring within a string.

Syntax

```
INSTR('http://www.infor.com', 'www')
```

Result: 8

LENGTH

Use LENGTH to return the length, or size, of a string expression. An empty string returns 0 length.

Syntax

```
select LENGTH(string_expression)
```

LPAD

Use LPAD to pad the left of a string with values up to an overall string length. The first parameter are the padding characters; enclose the characters with single quotes. The second parameter is the string expression. The third parameter is the overall length.

Syntax

```
select LPAD('padding_characters',string_expression, overall_length)
```

RPAD

Use RPAD to pad the right of a string with values up to an overall string length. The first parameter are the padding characters; enclose the characters with single quotes. The second parameter is the string expression. The third parameter is the overall length.

Syntax

```
select RPAD('padding_characters',string_expression, overall_length)
```

SUBSTR

Use SUBSTR to return a substring of a string. The first parameter is the string expression. The second parameter is starting position for the substring. The third parameter is the number of characters for the substring. The third parameter is optional.

Syntax

```
select SUBSTR(string_expression, start_position, number_of_characters)
```

Comparison functions

GREATEST

Use GREATEST to return the highest value from a list of parameters.

Syntax

```
select GREATEST(parameter1, parameter2, parameter3)
```

Infor functionality

Data Lake queries can contain keywords and functions proprietary to Infor. The functionality is useful to retrieve specific data from Data Lake, such as incremental load data. It is also used to retrieve localized string values from data objects.

Additional functionality allows you to view specific data object properties, such as the Data Lake data object ID, which allows you to trace a record back to a specific Data Lake payload object.

Queries for localized data

The locale keyword is used to select localized string values stored in data objects in the Data Lake.

The locale keyword is used to select localized string values stored in data objects in the Data Lake. For information on localized data, see “Data Selections for Localized String Data”. For Compass queries, the localized data is stored in Compass data storage when a query is run. Therefore, it is critical that the Locale Selections are defined before you query the data object.

If the Data Catalog locale selections are updated after data is stored in Compass data storage, use the administration stored procedure to clear the table and data. The next time a query is run, Compass data storage uses the current locale selections and includes the localized values stored in the Data Lake.

LOCALE keyword

Syntax

```
select locale(property,number of positions) from dataobject
```

Example

The localized property name is “greeting”, and the localized string is “hello” in en_US, “bonjour” in fr_CA and “hola” in Spanish. The Data Catalog locale selections are position 1, en_US, position 2, fr_CA and position 3, es_ES.

```
select locale(greeting,3) from dataobject
```

Results

```
greeting_1, greeting_2, greeting_3
```

```
hello, bonjour, hola
```

The locale keyword is case-insensitive. If the property name is not distinct, prefix the property with the data object or alias name, such as locale(product.name, 5). The number of positions defines the number of numbered columns in the result. The locale keyword works in conjunction with the Data Catalog Locale Selections.

You can also query the data object to show all the localized strings using the locale codes as the suffix of the column names.

You can also view the localized strings with the locale suffixes using a `select *` query. To achieve this, you can use the `includeInSelectAll` query hint in combination with the `L` parameter.

See Query hints on page 164.

If the Data Catalog Locale Selections change, it may be necessary to clear the data object's Compass data storage and Compass object definitions. When new locales are added to the Data Catalog, such as Portuguese and Italian, clear the object definition using the Infor Clear Table stored procedure. This process clears the object definition that includes the original locales but not the new locales of Portuguese and Italian.

It may also be necessary to clear the Compass data. If historic objects contain localized strings for Portuguese and Italian, clear the data. To clear the Compass data, use the Infor Clear Table with the option to clear reformatted data. Alternatively, use the Infor Clear Data option to clear Compass data. The next time a query runs, the Portuguese and Italian strings in the data are populated.

Alternatively, if the historic data does not include Portuguese and Italian strings, it does not have to be cleared. Queries for historic data have NULL values for Portuguese and Italian.

Data Catalog locale selections

Select the Data Catalog locale selections to associate the localized string position numbers with localized string values. The query returns the locale codes defined the Data Catalog Locale Selections function.

Syntax

```
select * from datacatalog.locale_selection
```

Queries for incremental data loads

The Data Lake contains every version, or variation, of a record. Each record in Compass data storage is associated with the Data Lake object ID and the Data Lake object datetime. The object datetime is the timestamp in which the data object "payload" was added to the Data Lake. The timestamp is a property of the payload object itself; it is not a property in the object or defined the data object metadata. The timestamp is referred to as the lastmodified property.

The lastmodified value is commonly used for incremental loads, so a full result set is not retrieved each time a query is run. For example, you can use a lastmodified value in a WHERE clause to retrieve data from data objects loaded on or after a specific datetime, such as 2019-09-01T09:30:47.323Z. Note that all lastmodified timestamps are in UTC in ISO8601 RFC3339 format with three fractional seconds.

The lastmodified value is available through two methods:

- You can use the `infor.lastmodifed()` function to select the lastmodified value. The parameter for the function is the data object or alias that is referenced in the query.

- You may add the lastmodified value to a query as a synthetic property. The property is synthetic because it is not defined in a data object's metadata.

Note: The `infor.lastmodified()` function is recommended over the lastmodified synthetic property. We recommend the function because it avoids contention with any data object that may have an actual lastmodified property.

You can also use the lastmodified synthetic property or the `infor.lastmodified()` function in the SELECT clause of the query to select the lastmodified value for each record.

You may also reference the value in the WHERE clause, GROUP BY, HAVING, and ORDER BY clauses, and in datetime functions.

Select the last modified value using the INFOR.LASTMODIFIED() function

Use the `infor.lastmodified()` function in the select clause of a query to return the timestamp that a record was added to the Data Lake. If the query includes more data objects, such as a query with a join, use the `infor.lastmodified('dataobject or alias')` function with a parameter that specifies the dataobject or alias on which the lastmodified value is based.

Note that the `infor.lastmodified()` function is case-insensitive. The function parameter is the data object or alias referenced in the query. If the query selects data from only one data object, the parameter is optional. If the query selects data from more than one data object or alias, the parameter is required. The data object or alias parameter must be enclosed with single quotes.

Use an alias for the function to distinguish the value in the result.

Example:

```
select property1, property2, infor.lastmodified() as dataobject_lastmodified
from dataobject
```

As an alternative, use the object name or alias as a parameter. The parameter must be enclosed within single quotes.

Example:

```
select property1, property2, infor.lastmodified('dataobject') as dataob
ject_lastmodified from dataobject
```

Result: the dataobject_lastmodified value is the Data Lake document timestamp in UTC.

Examples for a query using multiple data objects:

```
select do1.property1, do2.property5, infor.lastmodified('dataobject1') as
dataobject1_lastmodified from dataobject1 do1 inner join dataobject2 do2 on
do1.property1=do2.property1
```

Alternate method using the data object alias:

```
select do1.property1, do2.property5, infor.lastmodified('do1') as dataob
ject1_lastmodified from dataobject1 do1 inner join dataobject2 do2 on
do1.property1=do2.property1
```

Result: the last modified value is based on the record in dataobject1.

Select the last modified value using the LASTMODIFIED property

Select the lastmodified value in query to return the timestamp that a record was added to the Data Lake. If the query includes more data objects, such as a query with a join, use the lastmodified with a table or alias prefix on which the lastmodified value is based.

Note that the lastmodified property is case-insensitive. If the query combines data from more than one data object, the lastmodified property must be prefixed with the data object or alias.

Note that using the `infor.lastmodified()` function is the recommended method because the lastModified name may conflict with properties in the data object.

Example: `select property1, property2, lastmodified from dataobject`

Result: the lastmodified value is the Data Lake document timestamp in UTC.

Example for a query using multiple data objects:

```
select do1.property1, do2.property5, dataobject1.lastmodified from dataob
ject1 do1 inner join dataobject2 do2 on do1.property1=do2.property1
```

Result: the lastmodified value is based on the record in dataobject1.

Use the INFOR.LASTMODIFIED() function in a WHERE clause for an incremental data load

Use the `infor.lastmodified()` function in the WHERE clause of a query to filter results to a subset of records. The `infor.lastmodified()` function may be used for incremental data loads, to select only data that was loaded to the Data Lake since the previous update date.

The `infor.lastmodified()` function supports these operators:

- =
- >=
- <=
- BETWEEN

The timestamp value in a WHERE clause may be a timestamp in UTC in ISO8601 format with three milliseconds: `YYYY-MM-DDThh:mm:ss.sssZ`. It may also be a date or a timestamp that does not include milliseconds.

Example: `select property1,property2 from dataobject where infor.lastmodi fied()>= '2019-09-01T09:30:47.434Z'`

Example for a query using multiple data objects:

Example: `select do1.property1, do2.property5, do1.lastmodified from dataobject1 do1 inner join dataobject2 do2 on do1.property1=do2.property1 where in for.lastmodified('do2') >='2019-09-01T09:30:47.434Z'`

Query number, date and timestamp general and unparseable values

Compass data for numbers, including integers and big integers, dates and timestamps are stored in a general format for Compass queries. The general format is designed to be more accommodating of

decimal sizes and allowable values. This allows for Data Catalog data object metadata changes that do not require data to be reloaded into Compass data storage.

Data Catalog metadata for integers, big integers and decimals are all stored in Compass data storage as decimal(38,15). For example, if the Data Catalog metadata definition for a “quantity” is initially defined as a data type of integer. Values in the payload, though, include decimal values. The decimal values are stored in the general format of decimal(38,15). Later, the metadata definition is updated to a number data type with 2 decimal places. The Compass data does not have to be cleared and reloaded since the data is already stored as a decimal. A query for an integer value that has a decimal value in the Data Lake results in a NULL value. If the value is used in an aggregation or numeric function, the general data type is used.

Dates and timestamps, in the default format and Data Catalog available formats, are stored as timestamps in UTC

When data is converted to Compass data storage, if the general data type matches the payload values, the data is stored in the “general” format columns. If the data cannot be stored as a number or datetime, the value is unparseable. The unparseable value is stored in Compass as a string.

INFOR.SHOW_VALUE

The `infor.show_value` function is used to display the general values. If the value cannot be stored in the general format, you can query for the original value that could not be resolved as a number or timestamp.

Syntax:

```
select infor.show_value(property, 'general')
select infor.show_value(property, 'original')
```

The first parameter is the property name. Use double-quotes around property names that begin with digits or include spaces.

The second parameter is general or original. General format displays the general format of the value as decimal(38,15). Original value displays a string value. If the actual value from the payload is a number, date or timestamp and is stored in the general format, it does not appear on a request for the original format. Only values that could not be stored in the general format are stored in the original string format.

Infor functions

INFOR.CONCAT2 through INFOR.CONCAT6

infor.concat2, infor.concat3, infor.concat4, infor.concat5 and infor.concat6 allow you to concatenate two to six properties, literals or expressions together using a separator value. The number at the end of the function name determines the number of values that are concatenanted.

The first parameter is the separator character string. Use single quotes around the value to signify a string literal.

The second parameter determines if the separator is used after the first value if the subsequent value in the string is NULL. Specify Y or y if the separator is used when the following string is NULL.

The remaining parameters are the properties, literals or expressions to concatenate. The concatenation values may be strings. If an integer property is used, the value is converted to a string. If a date is

used, a datetime value is displayed in the results. Therefore, cast a date to a string to eliminate the time component. If a Boolean value is used, the result is true or false.

Syntax:

```
select infor.concat2('string','y',property1,property2)from dataobject
select infor.concat3('string','y',property1,property2,property3) from dataobject
select infor.concat4('string','y',property1,property2,property3,property4) from dataobject
select infor.concat5('string','y',property1,property2,property3,property4,property5) from dataobject
select infor.concat6('string','y',property1,property2,property3,property4,property5,property6) from dataobject
```

Example:

```
select infor.concat2('-','y',productid,productname) as productid_name from products
```

Result:

```
productid_name
12345-Pencil
23456-Pen
34567-Marker
```

```
select infor.concat6(',','y',street_address, apartment_suite,city,state,country) as address from customer_address
```

Result:

```
address
1250 Sunrise Place, 410, Orlando, Florida, USA
5067 Main Street, Orlando, Florida, USA
4325 Atlantic Avenue, 30B, Orlando, Florida, USA
```

In this example notice that the comma separator between street and city is not used if the apartment_suite value is null.

INFOR.DATETIME_TO_TIMESTAMP

The infor.datetime_to_timestamp function allows you to add hours and minutes to a date or timestamp value. The result is a timestamp. Note that the result format may vary depending on the query tool. All Compass queries return values in UTC time.

Syntax:

```
select infor.datetime_to_timestamp(datetimeproperty, 'hhmm')from dataobject
```

Example:

```
select orderdate,infor.datetime_to_timestamp(orderdate,'0230') as updated_orderdate from orders
```

Result:

```
orderdate, updated_orderdate
```

```
2019-07-13T08:00:00.000Z, 2019-07-13T10:30:00.000Z
```

The first parameter is the date or timestamp value. Use single quotes around the value to signify a string literal.

The second parameter value of hhmm is the hours and minutes to add to the datetime. The hour range is 00-24, and the minute range is 00-59. If the parameter is 2400, the date part of the first parameter is used and the time is set to 23:59:59.999.

INFOR.TIMESTAMP_TO_STRING

The infor.timestamp_to_string function allows you return all or part of a timestamp as a string. You may select to return only the date, the datetime or the full datetime with fractional seconds. The result is a string with no separator characters.

Syntax:

```
select infor.timestamp_to_string(datetime_expression,'Date')from dataobject
select infor.timestamp_to_string(datetime_expression,'Time')from dataobject
select infor.timestamp_to_string(datetime_expression,'Full')from dataobject
```

The first parameter is the timestamp value.

The second parameter value is Date to return a date as a string, The Time parameter returns the date and time to the second. The Full parameter returns the datetime with fractional seconds. Note that Date, Time and Full parameters are case-sensitive and must be enclosed within single quotes.

INFOR.STRING_TO_TIMESTAMP

The infor.string_to_timestamp function uses a date and adds hours, minutes, seconds and fractional seconds to it. The result is a timestamp. The third parameter is the level of granularity for the result, for example, at the hour, minute or second level. The fractional seconds are not used.

Syntax:

```
select infor.string_to_timestamp(dateproperty,'hhmmssf')from dataobject
```

Example:

```
select orderdate,infor.string_to_timestamp(orderdate,'0800000','Hours') as
datewithhours, infor.string_to_timestamp(orderdate,'043000','Minutes') as
datewithminutes,infor.string_to_timestamp(orderdate,'03333333','Seconds')
datewithseconds from orders
```

Result:

```
orderdate, datewithhours,datewithminutes,datewithseconds
2019-07-13, 2019-07-13 08:00:00,2019-07-13-04:30:00, 2019-07-13-03:33:33.000
```

The first parameter is the date value.

The second parameter value are the hours, minute and seconds to add to the date. The format is ‘hhmmssf’ where hh are the hours, mm are the minutes and ss are the seconds. Fractional seconds are not used. Enclose the value in single quotes.

The third parameter is the level of granularity of the timestamp result. The options are Hours, Minutes or Seconds. The values are case-sensitive and enclosed with single quotes. ‘Hours’ is based on the hh value. Values for mmssf are not used. ‘Minutes’ is based on the hhmm value. Values for ssf are not used. ‘Seconds’ is based on the hhmmss value. Fractional seconds are not used.

INFOR.SPLIT_VALUE

The infor.split_value function returns a portion of a string, based on a separator within the string. It is used to return a component of a string. For example, a multi-part customer account value is 001-324-26358213. Use the split value function to return only the third component, 26358213. The first parameter is the separator to locate in the string. The second parameter is the component. Component numbers 0 and 1 both return the first component of the string. The third parameter is the string.string to return based on the position of the separator in the string.

Syntax

```
infor.split_value('separator',compontent,string_expression)
```

Example

```
infor.split_value('-',3,'001-324-26358213')
```

Result

```
26358213
```

Query hints

Hints provide extra instructions to execute a query. Hints must be placed on the first lines of the query, before the SELECT or WITH clause. Query hints are optional, and one or more hints may be used. The hint syntax starts with two hyphens followed by an asterisk: –*.Hints are case insensitive, but they are documented with mixed case to distinguish the hint syntax.

includeAllVariations

This hint is used to return all variations of a record from the Data Lake.

For details and examples, see Variation handling on page 166.

The hint may be used to specify one or more data objects used in the query. Alternatively, you may use object aliases if the object is referenced more than once in a query.

Syntax

```
--*includeAllVariations=<<dataobject or alias>>/<<dataobject or alias>>
```

Add the hint before the first line of the query. The syntax is `--*includeAllVariations` and is case-insensitive. Do not use spaces or other characters in a hint. After the = , specify one or more data objects or aliases used in the query. Separate each value with a forward slash / . A data object reference applies to all data object references in the query, regardless of aliases. An alias reference applies the hint logic to the specific data object alias references only. Any data objects or alias that are not referenced in a hint, by default, return the maximum variation of each record, excluding records in which the maximum variation is deleted.

includeDeletionsWithMaxVariations

This hint is used to return the maximum, or highest, variations of a record from the Data Lake, regardless of the deletion indicator status.

For details and examples, see Variation handling on page 166.

The hint may be used to specify one or more data objects used in the query. Alternatively, you may use object aliases if the object is referenced more than once in a query.

Syntax

```
--*includeDeletionsWithMaxVariations=<<data object or alias>>/<<data object
or alias>>
```

Add the hint before the first line of the query. The syntax is --*includeDeletionsWithMaxVariation and is case-insensitive. Do not use spaces or other characters in a hint. After the = , specify one or more data objects or aliases used in the query. Separate each value with a forward slash / . A data object reference applies to all data object references in the query, regardless of aliases. An alias reference applies the hint logic to the specific data object alias references only. Any data objects or alias that are not referenced in a hint, by default, return the maximum variation of each record, excluding records in which the maximum variation is deleted.

includeInSelectAll

This hint adds additional properties to the results of a SELECT * query. The hint may include one or more parameters: S for synthetic properties, G for generated properties, P for the data storage partition and L for localized string values.

Parameters

- S for Synthetic properties

  Synthetic properties are properties associated with the Data Lake “payload” object. The payload object is the object stored in the Data Lake. The properties are lastModified and dataObjectId.

  - lastModified

    lastModified is the timestamp in which the data object was added to the Data Lake.

  - dataObjectId

    dataObjectId is the unique identifier of the Data Lake object from which the record is stored.

  The lastModified and dataObjectId are associated with each record and are used for traceability back to a specific object stored in the Data Lake. Use the dataObjectId to find a specific data object by using the Data Lake API streambyID. Each payload may contain one or more records, or tuples. Therefore, you may have the same dataObjectId and lastModified associated with more than one record in a result set.

- G for Generated properties

  Generated properties are properties added by Compass query services. The only generated property is autogenerated_timestamp.

  autogenerated_timestamp

  autogenerated_timestamp denotes the time in which the record was converted and stored for Data Lake queries.

- P for the Partition property

  The Partition property is the partition, or segment, of the Compass query data storage in which the record is stored. The partition column name is rd.

- L for Localized properties

  Localized properties are localized string values. Using the parameter returns the results for localized string data stored in the payload objects. Note that each localized string has a locale code suffix. The locale codes included match the locale codes defined in the Data Catalog Locale Selections.

  See Queries for localized data on page 157.

Use one or more parameters and separate each parameter with a forward slash /.

Syntax

```
--*includeInSelectAll=S/G/P/L
```

Example

```
--*includeInSelectAll=S/G/P
select * from products
```

Results: The results include all properties from products, and they include the additional properties:

```
. . . ,lastmodified, dataobjectid, autogenerated_timestamp, rd
2019-08-09T10:30:23.232Z, 8d963652-4a02-4c5a-bd37-26872593872c, 12, 2019-
08-09T11:13:07.317Z,2019-08-09
```

Example for localized strings

```
--*includeInSelectAll=L
select * from salutations
```

Results include all localized string properties. In the example, the property greeting is a localized string, and the locales are US-English, French Canadian and Spanish:

```
greeting_en_us, greeting_fr_ca, greeting_es_es hello, bonjour, hola
```

skipReformatting

This hint bypasses the process to convert raw objects to the Compass query format before executing the query. This hint is useful to verify the query syntax or to view data that has already been converted.

Note: Do not use this hint to retrieve a full result set, because the results do not include any new data loaded to the Data Lake.

The `skipReformatting` hint does not include parameters. The hint skips the data conversion process for all data objects referenced in the query.

Syntax

```
--*skipReformatting
```

Variation handling

Variation handling is based on the Data Object's metadata properties, defined in the additional properties for identifierpaths, variationpath and deleteindicator.

The identifierpaths property defines the property or properties, that comprise the primary key of a data object. For example, a Products data object may have Company and ProductID defined as the identifierpath properties, to signify that each product has a distinct company and product ID.

The variationpath is the property that defines a sequence structure for lower and higher variations. Lower variations signify earlier states, or versions of a record, and higher variations signify changes as updates are made to the record. Variations are generally integers. For example, the first version of a product is 1 when the order is created, 2 when the product is updated, 3 when the product is closed. Any update to the product triggers a new record, or variation, to be sent to the Data Lake.

The deleteindicator is a property that signifies that the record is physically or logically deleted from the source. For example, if a product is deleted, the deleted flag is set to true.

Example: Products

The example illustrates product records stored in the Data Lake. The identifier path, or primary key, of the records is Company+ProductID. The Variation column is the variation, or version, associated with each primary key. Notice that the Data Lake stores multiple variations, or versions, of each record, indicating the historic and current records. The DeletedFlag value of true or false indicates that a record was physically or logically deleted from the source, but the record exists in the Data Lake.

Company IdentifierPath	ProductID IdentifierPath	Description	Price	Variation VariationPath	DeletedFlag DeleteIndicator
001	991041	Bike	300	1	false
001	991041	Bike	350	2	true
002	222333	Scooter	129	1	false
002	222333	Scooter	132	2	false
003	12345	Skateboard	175	1	false
003	12345	Skateboard	195	2	true
003	12345	Skateboard	205	3	false

There are three methods to query Data Lake data for variations.

- Select the maximum variation of each record and exclude deleted records.
- Select all variations of each record.
- Select the maximum variation of each record and include records in which the maximum variation is deleted.

Select the maximum variation of each record and excluded deleted records

This logic matches most transaction systems. If a record is deleted, then the references to that record are physically or logically deleted. The purpose of this logic is to make the Data Lake appear as a regular transaction database, where records that are physically or logically deleted do not appear in query results.

This is the default behavior for all queries, so no query hint is required.

Query:

```
select * from Products
```

Result:

Company IdentifierPath	ProductID IdentifierPath	Description	Price	Variation VariationPath	DeletedFlag DeleteIndicator
002	222333	Scooter	132	2	false

Company IdentifierPath	ProductID IdentifierPath	Description	Price	Variation VariationPath	DeletedFlag DeleteIndicator
003	12345	Skateboard	205	3	false

The results include highest variation, variation 2, of record the for company 002, productID 222333, and the highest variation, variation 3, of the record for company 003, productId 12345. The record for company 001, product 991041 is not in the results because the maximum variation, variation 2, has a deleted status.

Select all variations of each record

Select all variations of records, regardless of the deleted indicator status. This logic is used to return historic and current records, and it may be used for auditing purposes or incremental data loads. The logic is useful to find a specific variation of a record, such as the version of the record when the record was created or updated to a specific status.

Select all variations

Use a query hint to return all variations for a data object or an alias reference in a query.

For details about the `includeAllVariations` hint, see Query hints on page 164.

Query:

```
--*includeallvariations=products

select * from Products
```

Company IdentifierPath	ProductID IdentifierPath	Description	Price	Variation VariationPath	DeletedFlag DeleteIndicator
001	991041	Bike	300	1	false
001	991041	Bike	350	2	true
002	222333	Scooter	129	1	false
002	222333	Scooter	132	2	false
003	12345	Skateboard	175	1	false
003	12345	Skateboard	195	2	true
003	12345	Skateboard	205	3	false

The results include all variations of all records, regardless of status.

Select the maximum variation of each record and include deleted records

This logic returns the maximum variation of each record, regardless of the deleted indicator. The highest variation of a record is returned, even if the highest variation record status is deleted. This logic is useful when the query joins tables, and a record may be deleted in one table but referenced in another.

Select all variations and include deleted records

Use a query hint to return the maximum variations including deleted records, for a data object or alias refrenced in a query.

For details about the `includeDeletionsWithMaximumVariations` hint, see Query hints on page 164.

Query:

```
--*includeDeletionsWithMaxVariations=products

select * from Products
```

Company IdentifierPath	ProductID IdentifierPath	Description	Price	Variation VariationPath	DeletedFlag DeleteIndicator
001	991041	Bike	350	2	true
002	222333	Scooter	132	2	false
003	12345	Skateboard	205	3	false

The results include highest variation of each record, regardless of the deleted status. The results include variation 2 of the record for company 001, productID 991041, variation 2 of the record for company 002, productID 222333, and the highest variation, variation 3, of the record for company 003, productID 12345.

Data Lake database schemas

The Data Lake incorporates schemas to separate data sets.

- Default

  The Data Lake data schema is Default. All the data objects stored in the Data Lake, and registered in the Data Catalog, will be retrieved from the Default schema.

- DataCatalog

  The Data Catalog schema contains Data Catalog data. Currently, the Data Catalog contains the Locale Selections, which are defined in the Data Catalog, Locale Selections function. Use the DataCatalog Locale Selections query to retrieve the locale codes, positions and substitute locales.

  See Queries for localized data on page 157.

- Information_Schema

The Information_Schema schema contains the schemata, tables and columns for Data Lake objects. Use the following queries to retrieve values.

Syntax:

```
select * from information_schema.schemata
```

Syntax:

```
select * from information_schema.tables
```

To retrieve a list of properties for a data object. Note that the columns data is dependent on Compass query activity. If a data object has not been referenced in a query, the column metadata is not included in the results.

```
select * from information_schema.columns where table_schema='default' and
table_name='dataobject'
```

To retrieve a list of data objects registered in the Data Catalog:

```
select * from information_schema.tables where table_schema='default'
```

Query processing

Compass queries are processed through a series of steps before the query returns results. The primary query steps are converting, or transforming, the Data Lake data and executing the query.

The data conversion steps read Data Lake data objects, also called “payloads”, data and convert the data to store it more efficiently for query purposes. This process involves retrieving the Data Lake DSV and newline-delimited JSON object data and converting it. The process also segments the data into partitions. The partitions divide the data, making it more efficient for a query to read. The partitioning method groups the data by the Data Lake addition date, referenced by the lastmodified property. This is the date that the data object is added to the Data Lake. This partitioning scheme is relevant for incremental data loads.

The Data Catalog object metadata is critical to the data conversion process. The object metadata provides the instructions used to process the data object schema into a table and column structure used by the queries. Several components of the metadata definition that are used are noted in the query considerations and best practices.

When Data Catalog data object definitions are updated, it may be necessary to clear the current data storage definition, the Compass data storage or both.

See Handling Data Catalog object metadata changes and Data Lake changes in Compass data through administration stored procedures on page 171.

Data objects are converted on-demand when a query is run. The data objects referenced in the query are compared to the latest data in the Data Lake. Any objects stored in the Data Lake that have not been converted, are converted when the query executes, to ensure that the query retrieves all data available for the query. A query will fail for a data conversion error.

For information on how to handle data conversion failures, see Query error handling on page 175.

The query execution steps involve preparing the query, executing the query and preparing the results.

To prepare the query, the data objects and properties referenced in the query are validated against the Data Catalog object metadata, the query hints and syntax are validated and converted, the query is executed and the results are prepared.

For information on how to handle errors in the query execution steps, see Query error handling on page 175.

Handling Data Catalog object metadata changes and Data Lake changes in Compass data through administration stored procedures

The Infor Data Catalog is the source of metadata definitions for Compass data objects, data object properties, and logic for variation handling. When you make metadata changes, use the administration stored procedures to clear outdated metadata definitions.

You can update the Data Lake data through the purge and archive processes and the process to mark objects as corrupt.

Compass is not automatically updated when object metadata is updated or when Data Lake data is purged, archived, or marked as corrupt.

Use the Compass stored procedures to perform administrative operations on Compass data storage, object definitions, and object views. You use the stored procedure to clear table, clear data, or reset data when Data Lake data is purged, archived, or marked as corrupt. Use the clear table or clear view procedures to update Compass for object metadata changes. You use the reset partitions stored procedure to resolve partition issues.

The administration stored procedure to clear table removes the Compass object definition and variation handling views. The clear table can clear table data object definitions with or without removing converted data. The data and object definitions are recreated the next time a query is processed for the affected data objects.

Data Catalog object metadata best practices

It is expected that the Data Catalog object definitions change over time. Use the best practice guidelines to ensure that object definitions are in synch with Compass data storage and query functionality.

- Add new properties to the metadata as required.
- Add a new property to replace an existing property, leave the deprecated property in the metadata definition.
- Loosen constraints of existing properties, such as extending string lengths or increasing the scale and precision of decimal values. Do not tighten constraints.
- Do not remove or rename properties.
- Do not change the data type of an existing property. This makes old and new data incompatible. Instead, add a new property.

- Use the identifier paths, variation path and delete indicator properties for variation handling.
  See Variation handling on page 166.

Clear table with or without clearing data

The infor.clear_table stored procedure clears the Compass object definition. The Compass object definition includes the properties in the data object plus information about the object's identifier paths, variation path and deleted indicator. Optionally, the stored procedure clears the Compass data storage.

This procedure is used to clear Compass when the object's metadata definition, stored in the Data Catalog, is updated with changes that affect the core definition or when historic data must be cleared and stored again with the new definition. This procedure does not affect data in the "raw" Data Lake; it updates data stored for Compass data storage only.

This procedure is used when Data Lake data is purged, archived, or marked as corrupt. This operation clears all Compass data storage; it does not affect the "raw" Data Lake data.

Use this procedure when these changes occur:

- The data object metadata definition is updated with significant data type changes, such as changing a data type from a string to a number, or a number to a datetime. If Data Lake data has already been converted to Compass data storage, clear the table and the data.
- The data object metadata definition is updated with new properties. Use a value of 'true' to reload historic data based on the updated object definition. The data is refreshed when a query is executed on the data object.
- The data object metadata definition is updated with new properties. Use a value of 'false' to not clear the data if historic data is not affected and does not have to be converted again to Compass data storage.
- The Data Catalog Locale Selections change. Clear the table definitions when new locales are added or updated. The next time a query runs, the object is converted using the current Locale Selections. If historic data contains localized strings for the new or updated locale codes, clear the Compass data storage. If historic data, stored in Compass data storage does not contain localized strings for the new or updated locales, clearing the data is not necessary.
- The Data Lake data is purged, archived, or marked as corrupt.

INFOR.CLEAR_TABLE and clear Compass data

The first parameter is the object name. Enclose the object name in single quotes. The second parameter is true or false. It is a string literal and must be enclosed within single quotes. A value of true clears data stored for Compass queries. Query syntax is case-insensitive.

Syntax

```
EXEC INFOR.CLEAR_TABLE('objectname', 'true')
```

INFOR.CLEAR_TABLE and retain Compass data

The infor.clear_table stored procedure clears the optimized data format storage definition. Use the second parameter option of 'false' to retain the historic data stored in Compass. The first parameter is the object name. Enclose the object name in single quotes. The second parameter is true or false. It is a string literal and must be enclosed within single quotes. A value of false retains Compass data. Query syntax is case-insensitive.

Syntax

```
EXEC INFOR.CLEAR_TABLE('objectname','false')
```

Clear view

The infor.clear_view stored procedure clears the Compass view definition for a data object. Use this procedure when the Data Catalog object definition's additional properties for IdentifierPaths, VariationPath or DeleteIndicator properties are added, updated or removed. The values impact the variation handling query processes.

INFOR.CLEAR_VIEW

The parameter is the object name. Enclose the object name in single quotes. Query syntax is case-insensitive.

Syntax

```
EXEC INFOR.CLEAR_VIEW('objectname')
```

Clear data

The infor.clear_data stored procedure clears the Compass data stored for an object.

Use this process when the Data Catalog object definition changes, that requires clearing Compass data and reconverting it again based on the new metadata definition.

Use the infor.clear_data stored procedure to clear Compass data after Data Lake data is purged, archived, or marked as corrupt. The date on the clear data operation should be set back to the earliest date, in UTC, of the Data Lake operation. For example, if you purged data from 2020-01-01, use the clear data operation to clear data from 2020-01-01. The next time you run a Compass query, the Compass data is reprocessed from 2020-01-01 to the current time. Compass data that was added to the Data Lake before 2020-01-01 is not affected.

The second parameter is the partition of data. Partitions are based on UTC date. The data cleared is data posted on and after the date. This procedure does not affect data in the "raw" Data Lake; it updates data stored for Compass data storage only.

INFOR.CLEAR_DATA

The first parameter is the object name. Enclose the object name in single quotes.

The second parameter is the Compass data partition. Partitions correspond to dates. The dates are based on the date of the last modified timestamp. The lastmodified timestamp is the date time on which the data object was added to the Data Lake. The clear data procedure clears all Compass data on and after the specified date. Enclose the date with single quotes. Query syntax is case-insensitive.

Syntax

```
EXEC INFOR.CLEAR_DATA('objectname', 'YYYY-MM-DD')
```

Reset partitions

The infor.reset_partitions stored procedure clears the Compass data partitions and recreates them. Use this process when Compass data is missing or incomplete. This condition may occur if converting “raw” data objects to Compass data failed.

This procedure does not affect Compass data definitions or data. It only impacts Compass data partitions.

INFOR.RESET_PARTITIONS

The parameter is the object name. Enclose the object name in single quotes. Query syntax is case-insensitive.

Syntax

```
EXEC INFOR.RESET_PARTITIONS('objectname')
```

Compass JDBC driver

The Compass JDBC driver may be used to query Data Lake data through a local SQL query tool.

The same query syntax is supported through the Compass query editor, the Compass APIs, and the Compass JDBC driver.

The Compass JDBC driver is available through the ION Desk Downloads function. You can access the Downloads function through ION Desk / Configurations / Downloads. See the *Infor ION Desk User Guide*.

The driver is also available for download through the Infor Support Download site.

The JDBC driver uses ION API authentication.

The setup and configuration instructions are documented in Infor Support Portal KB 2103864. We recommend that you use the current version of the driver because support for older versions is deprecated over time.

Query result set differences

In the Data Catalog, the metadata object’s properties have data types associated with them. For example, integer, decimal, Boolean or strings defined with a date or datetime format.

The result values may differ from the metadata-defined type. Query results are based upon the method that is used to execute a query. For the Compass API, the values are also defined by the output type of CSV or new-line-delimited JSON.

The Compass query editor shows results in a grid. If you export the results to CSV, the format values are set.

The Compass API returns results in CSV or JSON, depending on the result set format that is specified when the query is submitted. The Compass API result values that are returned for JSON and CSV output show values as strings using double-quotes. The CSV query results return numbers in the

general format of decimal (38,15). The CSV query results return dates and timestamps in UTC using the ISO8601 RFC3339 format. For CSV, the results do not convert the results to the metadata-defined data type, scale, and precision or date format. The result format and values may be changed in future versions of Compass queries. The JSON result format returns numeric values according to their metadata-defined type, when applicable. Integers and bigints are returned, and decimal values are based on the metadata-defined precision and scale.

Boolean values are returned as true or false.

Date and datetime values are returned as timestamps in UTC using the ISO8601 RFC3339 format.

The JDBC result format may also be converted based on the SQL query tool's format or localization settings. The JDBC result returns timestamps in UTC using the ISO8601 RFC3339 format. The query tool may convert the value to the local timezone. The query tool may convert the format. Check the query tool settings for data type controls and localization settings.

Query error handling

Compass queries use messages categories to distinguish which part, or component, failed during the query execution.

These are the message categories:

- SQL messages, for query errors that occur during the query process execution. The message code range for SQL messages is 200-399.
- Process messages, for query errors that occur because of invalid data object or property references and query conversion issues. The message code range for SQL messages is 400-499.
- Data storage messages, for queries that fail during the Compass data storage process. The message code range for data storage messages is 500-599.
- Internal messages, for queries that fail for general errors. The message code range for data storage messages is 600-699.
- Compass API messages. The message code range for API messages is 700-799.

Each query is associated with a unique query ID. The query ID is crucial to helping you and Infor Support investigate an error. A query ends on the first error. The error message includes the query ID, the timestamp in UTC, and other error details including the error category, error code and message. When reporting issues to Infor Support, include the full error message details, including the query ID, error timestamp and messages.

Troubleshooting Compass queries

Use these troubleshooting guidelines to answer common questions and issues.

Initial query is executing for a long time

The first query executed for data object or data objects referenced in a query initiates a Compass data storage activity. For Compass data storage, all data in the "raw" or original Data Lake that has not been

stored in the Compass data storage is converted. The Compass data storage starts before the query executes, so the Compass query executes on the most recent data in the Data Lake.

For example, 1000 data objects are sent to the Data Lake, and a query is executed for the data object. All 1000 objects are stored in the optimized Compass data storage before the query executes. This occurs regardless of the WHERE condition on a query. This condition only occurs one time; once the data is stored in the Compass data store, it stays in the store unless it is cleared using an administration stored procedure.

To minimize the effect of the data storage time, execute queries more frequently. In this way, fewer data objects must be stored in Compass storage each time a query is executed.

If you are testing query syntax or do not necessarily require the current Data Lake data, use the hint to “skip reformatting”. The hint skips the process to store Data Lake data into Compass data storage.

See Query hints on page 164.

Updated Data Catalog data object metadata changes are not in Compass

Data Catalog data object metadata changes are not automatically updated for historic data or for property changes, such as these:

- Adding properties to a metadata definition
- Updating additional properties for identifier paths, variation path, and the deletion indicator

These changes require clearing the Compass table and Compass data.

For guidelines and instructions to update Compass to match metadata definitions, see Handling Data Catalog object metadata changes and Data Lake changes in Compass data through administration stored procedures on page 171.

Updated Data Catalog locale selections cause queries to fail with a 401 error

Data Catalog locale selection changes are not automatically updated for historic data or for locale selection changes, such as adding new locales to locale selections.

These changes require clearing the Compass table and Compass data.

For guidelines and instructions to update Compass to match the locale selections, see Handling Data Catalog object metadata changes and Data Lake changes in Compass data through administration stored procedures on page 171.

Query results are incorrect when the query results have duplicate result column names

A query that selects properties or aliases with the same name returns only distinct properties in the results. As a general guideline, do not select properties or aliases with duplicate names. Ensure that each property in a result set has a distinct name.

Queries fail after object metadata is updated

A query may fail with various errors, including a 401 error for invalid property names, after Data Catalog object metadata is updated. Metadata changes are not automatically reflected in Compass. If object metadata is changed, clear the Compass object definition and clear Compass data if necessary.

See the INFOR.CLEAR_TABLE stored procedure in the section Handling Data Catalog object metadata changes and Data Lake changes in Compass data through administration stored procedures on page 171.

Appendix A: Valid characters for document names

Custom documents can use any standard letter from any language.

Upon import, the allowed characters are verified using a regular expression that is similar to this sample expression:

```
/[\u0041-\u005A\u0061-\u007A\u00AA\u00B5\u00BA\u00C0-\u00D6\u00D8-
\u00F6\u00F8-\u02C1\u02C6-\u02D1\u02E0-\u02E4\u02EC\u02EE\u0370-
\u0374\u0376\u0377\u037A-\u037D\u0386\u0388-\u038A\u038C\u038E-\u03A1\u03A3-
\u03F5\u03F7-\u0481\u048A-\u0527\u0531-\u0556\u0559\u0561-\u0587\u05D0-
\u05EA\u05F0-\u05F2\u0620-\u064A\u066E\u066F\u0671-
\u06D3\u06D5\u06E5\u06E6\u06EE\u06EF\u06FA-\u06FC\u06FF\u0710\u0712-
\u072F\u074D-\u07A5\u07B1\u07CA-\u07EA\u07F4\u07F5\u07FA\u0800-
\u0815\u081A\u0824\u0828\u0840-\u0858\u08A0\u08A2-\u08AC\u0904-
\u0939\u093D\u0950\u0958-\u0961\u0971-\u0977\u0979-\u097F\u0985-
\u098C\u098F\u0990\u0993-\u09A8\u09AA-\u09B0\u09B2\u09B6-
\u09B9\u09BD\u09CE\u09DC\u09DD\u09DF-\u09E1\u09F0\u09F1\u0A05-
\u0A0A\u0A0F\u0A10\u0A13-\u0A28\u0A2A-
\u0A30\u0A32\u0A33\u0A35\u0A36\u0A38\u0A39\u0A59-\u0A5C\u0A5E\u0A72-
\u0A74\u0A85-\u0A8D\u0A8F-\u0A91\u0A93-\u0AA8\u0AAA-\u0AB0\u0AB2\u0AB3\u0AB5-
\u0AB9\u0ABD\u0AD0\u0AE0\u0AE1\u0B05-\u0B0C\u0B0F\u0B10\u0B13-\u0B28\u0B2A-
\u0B30\u0B32\u0B33\u0B35-\u0B39\u0B3D\u0B5C\u0B5D\u0B5F-
\u0B61\u0B71\u0B83\u0B85-\u0B8A\u0B8E-\u0B90\u0B92-
\u0B95\u0B99\u0B9A\u0B9C\u0B9E\u0B9F\u0BA3\u0BA4\u0BA8-\u0BAA\u0BAE-
\u0BB9\u0BD0\u0C05-\u0C0C\u0C0E-\u0C10\u0C12-\u0C28\u0C2A-\u0C33\u0C35-
\u0C39\u0C3D\u0C58\u0C59\u0C60\u0C61\u0C85-\u0C8C\u0C8E-\u0C90\u0C92-
\u0CA8\u0CAA-\u0CB3\u0CB5-\u0CB9\u0CBD\u0CDE\u0CE0\u0CE1\u0CF1\u0CF2\u0D05-
\u0D0C\u0D0E-\u0D10\u0D12-\u0D3A\u0D3D\u0D4E\u0D60\u0D61\u0D7A-\u0D7F\u0D85-
\u0D96\u0D9A-\u0DB1\u0DB3-\u0DBB\u0DBD\u0DC0-\u0DC6\u0E01-
\u0E30\u0E32\u0E33\u0E40-
\u0E46\u0E81\u0E82\u0E84\u0E87\u0E88\u0E8A\u0E8D\u0E94-\u0E97\u0E99-
\u0E9F\u0EA1-\u0EA3\u0EA5\u0EA7\u0EAA\u0EAB\u0EAD-
\u0EB0\u0EB2\u0EB3\u0EBD\u0EC0-\u0EC4\u0EC6\u0EDC-\u0EDF\u0F00\u0F40-
\u0F47\u0F49-\u0F6C\u0F88-\u0F8C\u1000-\u102A\u103F\u1050-\u1055\u105A-
\u105D\u1061\u1065\u1066\u106E-\u1070\u1075-\u1081\u108E\u10A0-
\u10C5\u10C7\u10CD\u10D0-\u10FA\u10FC-\u1248\u124A-\u124D\u1250-
\u1256\u1258\u125A-\u125D\u1260-\u1288\u128A-\u128D\u1290-\u12B0\u12B2-
\u12B5\u12B8-\u12BE\u12C0\u12C2-\u12C5\u12C8-\u12D6\u12D8-\u1310\u1312-
\u1315\u1318-\u135A\u1380-\u138F\u13A0-\u13F4\u1401-\u166C\u166F-
```

```
\u167F\u1681-\u169A\u16A0-\u16EA\u1700-\u170C\u170E-\u1711\u1720-
\u1731\u1740-\u1751\u1760-\u176C\u176E-\u1770\u1780-\u17B3\u17D7\u17DC\u1820-
\u1877\u1880-\u18A8\u18AA\u18B0-\u18F5\u1900-\u191C\u1950-\u196D\u1970-
\u1974\u1980-\u19AB\u19C1-\u19C7\u1A00-\u1A16\u1A20-\u1A54\u1AA7\u1B05-
\u1B33\u1B45-\u1B4B\u1B83-\u1BA0\u1BAE\u1BAF\u1BBA-\u1BE5\u1C00-\u1C23\u1C4D-
\u1C4F\u1C5A-\u1C7D\u1CE9-\u1CEC\u1CEE-\u1CF1\u1CF5\u1CF6\u1D00-\u1DBF\u1E00-
\u1F15\u1F18-\u1F1D\u1F20-\u1F45\u1F48-\u1F4D\u1F50-
\u1F57\u1F59\u1F5B\u1F5D\u1F5F-\u1F7D\u1F80-\u1FB4\u1FB6-\u1FBC\u1FBE\u1FC2-
\u1FC4\u1FC6-\u1FCC\u1FD0-\u1FD3\u1FD6-\u1FDB\u1FE0-\u1FEC\u1FF2-
\u1FF4\u1FF6-\u1FFC\u2071\u207F\u2090-\u209C\u2102\u2107\u210A-
\u2113\u2115\u2119-\u211D\u2124\u2126\u2128\u212A-\u212D\u212F-\u2139\u213C-
\u213F\u2145-\u2149\u214E\u2183\u2184\u2C00-\u2C2E\u2C30-\u2C5E\u2C60-
\u2CE4\u2CEB-\u2CEE\u2CF2\u2CF3\u2D00-\u2D25\u2D27\u2D2D\u2D30-
\u2D67\u2D6F\u2D80-\u2D96\u2DA0-\u2DA6\u2DA8-\u2DAE\u2DB0-\u2DB6\u2DB8-
\u2DBE\u2DC0-\u2DC6\u2DC8-\u2DCE\u2DD0-\u2DD6\u2DD8-
\u2DDE\u2E2F\u3005\u3006\u3031-\u3035\u303B\u303C\u3041-\u3096\u309D-
\u309F\u30A1-\u30FA\u30FC-\u30FF\u3105-\u312D\u3131-\u318E\u31A0-
\u31BA\u31F0-\u31FF\u3400-\u4DB5\u4E00-\u9FCC\uA000-\uA48C\uA4D0-
\uA4FD\uA500-\uA60C\uA610-\uA61F\uA62A\uA62B\uA640-\uA66E\uA67F-\uA697\uA6A0-
\uA6E5\uA717-\uA71F\uA722-\uA788\uA78B-\uA78E\uA790-\uA793\uA7A0-
\uA7AA\uA7F8-\uA801\uA803-\uA805\uA807-\uA80A\uA80C-\uA822\uA840-
\uA873\uA882-\uA8B3\uA8F2-\uA8F7\uA8FB\uA90A-\uA925\uA930-\uA946\uA960-
\uA97C\uA984-\uA9B2\uA9CF\uAA00-\uAA28\uAA40-\uAA42\uAA44-\uAA4B\uAA60-
\uAA76\uAA7A\uAA80-\uAAAF\uAAB1\uAAB5\uAAB6\uAAB9-\uAABD\uAAC0\uAAC2\uAADB-
\uAADD\uAAE0-\uAAEA\uAAF2-\uAAF4\uAB01-\uAB06\uAB09-\uAB0E\uAB11-
\uAB16\uAB20-\uAB26\uAB28-\uAB2E\uABC0-\uABE2\uAC00-\uD7A3\uD7B0-
\uD7C6\uD7CB-\uD7FB\uF900-\uFA6D\uFA70-\uFAD9\uFB00-\uFB06\uFB13-
\uFB17\uFB1D\uFB1F-\uFB28\uFB2A-\uFB36\uFB38-
\uFB3C\uFB3E\uFB40\uFB41\uFB43\uFB44\uFB46-\uFBB1\uFBD3-\uFD3D\uFD50-
\uFD8F\uFD92-\uFDC7\uFDF0-\uFDFB\uFE70-\uFE74\uFE76-\uFEFC\uFF21-
\uFF3A\uFF41-\uFF5A\uFF66-\uFFBE\uFFC2-\uFFC7\uFFCA-\uFFCF\uFFD2-
\uFFD7\uFFDA-\uFFDC]+
```